GET MORE FROM THE VIC 20

OWEN BISHOP

Other books of interest from Granada:

The Apple II

APPLE II PROGRAMMERS HANDBOOK
R. C. Vile
0 246 12027 4

ATARI

GET MORE FROM THE ATARI
Ian Sinclair
0 246 12149 1

THE ATARI BOOK OF GAMES
M. James, S. M. Gee and K. Ewbank
0 246 12277 3

The BBC Micro

INTRODUCING THE BBC MICRO
Ian Sinclair
0 246 12146 7

THE BBC MICRO – AN EXPERT GUIDE
Mike James
0 246 12014 2

21 GAMES FOR THE BBC MICRO
M. James, S. M. Gee and K. Ewbank
0 246 12103 3

BBC MICRO GRAPHICS AND SOUND
Steve Money
0 246 12156 4

DISCOVERING BBC MICRO MACHINE CODE
A. P. Stephenson
0 246 12160 2

THE BBC BASIC PROGRAMMER
S. M. Gee and M. James
0 246 12158 0

BBC MICRO AND ELECTRON MACHINE CODE – AN EXPERT GUIDE
A. P. Stephenson and D. J. Stephenson
0 246 12227 7

The Colour Genie

MASTERING THE COLOUR GENIE
Ian Sinclair
0 246 12190 4

The Commodore 64

COMMODORE 64 COMPUTING
Ian Sinclair
0 246 12030 4

THE COMMODORE 64 GAMES BOOK
Owen Bishop
0 246 12258 7

SOFTWARE 64 Practical Programs for the Commodore 64
Owen Bishop
0 246 12266 8

The Dragon 32

THE DRAGON 32 And How to Make The Most Of It
Ian Sinclair
0 246 12114 9

THE DRAGON 32 BOOK OF GAMES
M. James, S. M. Gee and K. Ewbank
0 246 12102 5

THE DRAGON PROGRAMMER
S. M. Gee
0 246 12133 5

DRAGON GRAPHICS AND SOUND
Steve Money
0 246 12147 5

THE DRAGON 32 How to Use and Program
Ian Sinclair
0 586 06103 7

The Electron

THE ELECTRON PROGRAMMER
S. M. Gee and Mike James
0 246 12340 0

21 GAMES FOR THE ELECTRON
Mike James, S. M. Gee and K. Ewbank
0 246 12344 3

BBC MICRO AND ELECTRON MACHINE CODE – AN EXPERT GUIDE
A. P. Stephenson and D. J. Stephenson
0 246 12227 7

Learning is Fun! 40 EDUCATIONAL GAMES FOR THE BBC MICRO AND ELECTRON
Vince Apps
0 246 12317 6

The IBM Personal Computer

THE IBM PERSONAL COMPUTER
James Aitken
0 246 12151 3

The Jupiter Ace

THE JUPITER ACE
Owen Bishop
0 246 12197 1

The Lynx

LYNX COMPUTING
Ian Sinclair
0 246 12131 9

The NewBrain

THE NEWBRAIN And How To Make The Most Of It
Francis Samish
0 246 12232 3

The ORIC-1

THE ORIC-1 And How To Get The Most From It
Ian Sinclair
0 246 12130 0

THE ORIC-1 BOOK OF GAMES
M. James, S. M. Gee and K. Ewbank
0 246 12155 6

Continued on inside back cover

Get More From the VIC 20

Also by Owen Bishop
THE VIC 20 GAMES BOOK
21 Challenging Games

There is something for every member of the family in this book of twenty-one fascinating games. They have all been written to exploit in full the excellent graphics and sound capabilities of the VIC 20 and are fully tested and crash-proofed.

Each program is accompanied by instructions telling you how to play the game and often some tips on winning tactics. Hints are also given on how to adapt the game to suit your own ideas wherever possible.

ISBN 0 246 12187 4

Get More From the VIC 20

Owen Bishop

GRANADA
London Toronto Sydney New York

Granada Technical Books
Granada Publishing Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Granada Publishing 1983
Reprinted 1983

Copyright © Owen Bishop 1983

British Library Cataloguing in Publication Data
Bishop, O. N.
Get more from the VIC 20.
1. VIC 20 (Computer)—Programming
2. Basic (Computer program language)
I. Title
001.64'24 QA76.8.V5

ISBN 0-246-12148-3

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed in Great Britain by Mackays of Chatham, Kent

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the publishers.

Acknowledgement

We wish to thank Commodore Business Machines, Slough, for their advice and assistance in connection with the preparation of this book.

Contents

About This Book 2

1 VIC – The User-friendly Computer 3

2 Setting It Up 9

3 A Splash of Colour 21

4 Back to Basics 34

5 Sounds Fantastic! 50

6 The Organised Machine 69

7 More About Graphics 81

8 Sorting It Out 102

9 More About Sound 117

10 The Functioning Machine 128

11 Colour on the Move 138

12 The Organised Programmer 158

Appendix A – Useful Tables 170

Appendix B – Converting Programs to Larger RAM 182

Index 183

About This Book

The range of topics covered by this book has been chosen with three types of reader in mind. First, there are those who are beginning computing and who have just bought or are about to buy a VIC 20. The manual which is supplied with the VIC 20 is a good one for the earliest stages, but this book will take you on from there, and give you lots of fun along the way.

Then there are those readers who have come to VIC 20 computing from another computer. They already know something – perhaps quite a lot – about BASIC and want to apply their knowledge to the VIC. This book will familiarise you with VIC BASIC and show you how to use it effectively.

Finally, there are those who are attracted to the VIC by its emphasis on colour graphics and its versatile sound generators. This book will help you master the use of these facilities in an entertaining way.

The programs in this book have all been written and tested to run on an unexpanded (3.5 kilobyte) VIC 20. If you have more than 3 kilobytes of extra memory, it is necessary to alter certain values in a few of the programs, as explained in Appendix B.

Chapter One

VIC – The User-friendly Computer

There are lots of things you can do with a computer. Here are some of them:

Games
For one player, for two players, or for the whole family. In many of the single-player games, you play against the computer, improving your performance until you win.

Music
There is a lot of fun to be had with a music program. You can listen to the computer play, or you can use its keyboard to play tunes of your own. The VIC is particularly good with this kind of program.

Money
There are programs to help you manage your domestic budget. There are programs which keep business accounts and help in planning the future of your business.

Pictures
Drawing coloured pictures on the screen is a creative pastime. There are several programs available to help you get started on 'computer art'. The VIC is ideal for producing bold and colourful pictures.

Information
All kinds of information can be stored in a computer, perhaps sorted or analysed, or filed away ready to be retrieved whenever you want it. Use programs such as these for keeping an address-book, a record of your stamp collection, a list of the shows you have seen on TV, a compilation of your favourite recipes, or a data base of customer-information.

Education
There are dozens of ways in which computers can help you or your child to explore new subjects and learn about them.

Word processing
If you have a printer attached to your computer, you can use the computer as a word processor. This makes typing really easy. The computer can do things with words that no typewriter or typist can ever do. For home and business, you obtain a really professional printout with the minimum of effort or skill.

Programming
Last but not least, what is for many the most absorbing aspect of home computing – making the computer do exactly what you want it to do. The VIC is easier to program than most computers, and this book tells you just how to set about it.

Hardware and software

When you unpack the computer you will find several items in the box. The largest and most expensive is the computer itself. You will also find a power supply box, an RF modulator and various cables. Chapter Two deals with how to connect all these together. You also need some kind of video screen. Most people use their ordinary TV set for this, but it is possible to use a video screen specially designed for use with computers – usually called a *monitor*. More about this later, too. All these items together are known as the *hardware*.

You will have gathered from the descriptions above that the hardware does not do much on its own. Before it can play games or make music, or add up your accounts, it needs a program. Programs are often referred to as software. A program tells the computer what to do. Without a program it will do just nothing!

In fact, the computer already has one program in it when you buy it. This becomes active when you switch on, and tells the computer how to find out what keys are being pressed on the keyboard. It also tells it how to do other routine things such as make messages and pictures appear on the TV screen. In short, this is a program which tells the computer how to set about its essential routine activities. This program is often called the *monitor* program. This is a confusing name, since we have already used the word 'monitor' to refer to the specially-designed video screen. But since most people

use a TV instead of a monitor, and since most of the time we are hardly aware of what the monitor program is doing, the confusion is not one to worry about. At least when you come across the two names you will know which is which! Incidentally, since the monitor program is already built in to the machine and really is a firm part of it, some people call it the *firmware*.

The monitor program is kept in a part of the computer which we call its *memory*. It is held permanently in part of the memory (called ROM) which is specially designed for holding a fixed program. The computer can not 'forget' its monitor program when it is switched off. By contrast, the games programs and other software are held in a changeable part of memory, called RAM. Whatever program you want to use is put into that part of memory. It is held there indefinitely, but disappears if you replace it with another program or when you switch off the computer's power supply.

Where do you get the software to tell the VIC how to play all those games or do the other things listed at the beginning of the chapter? Software comes in a variety of different forms, which we can list according to the way you put them into the VIC's memory:

Keyboard

If you invent your own programs, you type them on the keyboard as you go along and they are automatically put into memory. You will also find lots of programs listed in popular computing magazines and in books. You simply copy them into the computer by typing them on the keyboard. When you have finished, just press the 'RETURN' key and the fun begins. By the time you have finished using this book you will be very much at home with the keyboard. As well as the short programs intended to help you understand how to control the VIC, this book lists some programs which will give you hours of amusement from your computer. There is also a companion book to this one, called *The VIC 20 Games Book*, which contains the programs for 21 exciting new games.

Tape

The VIC has its own special cassette recorder. You can buy software on cassettes, ready to be loaded into your VIC. There are dozens, if not hundreds, of different programs available on tape for the VIC.

Having programs on tape makes it easier and quicker to put them into the computer. There is no risk of typing errors either! You can also use the cassette recorder to store programs which you have keyed in from magazines or books, or programs which you have

written yourself. This is called 'saving' a program. When you save a program the contents of the computer's program memory (RAM) are recorded on the tape. This gives a permanent record of the program, which can be loaded back into the computer at any time in the future.

Cartridge
You can buy programs in cartridges which plug into a special slot in the VIC. In a way, this is rather like having the program in ROM, for the program is there instantly, as soon as you switch on the computer. It is permanent and cannot be lost. Cartridges are more expensive than tapes, but rather more convenient to use. Also, since each program is held in the memory of the cartridge, it does not use up any of the memory of the computer. Cartridges are especially useful for holding what are called *utility programs*, such as a word processor program or the 'Programmers' aid' programs. The program is in its own ROM, leaving the whole of the RAM free to hold the text which is being processed, or the program which is being written. There are also cartridges which do not contain any program at all but contain additional empty memory (RAM), for your longer programs.

Disk
These are also known as 'floppy disks' or 'diskettes'. Like the cassette tape, they store the program in magnetic form. You need to connect a disk drive to your VIC if you are using disks. Disk drives are fairly expensive but disks have all the advantages of cassette tapes, and more beside. Disks are more reliable than tapes, they hold many more programs and programs can be loaded into the computer or saved much faster than with cassettes. If you ever become a really serious programmer, or are using your computer for business (when 'time is money') you will probably decide to invest in a disk drive.

Getting started

You need very little to get started into the friendly and colourful world of VIC computing. The essential items are:

The VIC 20 colour computer
The cassette deck
The power supply

The RF modulator
The video cable (the other cables are already attached to one or other of the above items)

All of these items are usually supplied as a package deal. You also need a TV set, preferably a colour TV, though the VIC does work on monochrome sets, too. The VIC uses the sound channel of the TV set for its music and sound effects. This makes it different from many computers which have a loudspeaker inside the computer. The great advantage of the VIC's system is that you get sound of much better quality, and you have full control over its volume. You can make it really loud so as to enjoy the effects, or you can turn it down low while you are experimenting with the programming.

As an alternative to the TV set, you can use a video monitor, either a colour monitor or a monochrome one. A monitor will usually give a rather sharper picture than a TV set but, unless the monitor has a built-in sound channel, you will not be able to hear the VIC's sound effects. Before buying a monitor it is important to consult your dealer, to make certain that it is suitable for connecting to a VIC 20.

Some early models of black-and-white TV sets are not suited for connecting to a computer, but if your set is capable of receiving BBC2 and other stations on the UHF wavebands, it should be all right. If in doubt, consult your VIC dealer when buying the computer.

In addition to the hardware, you will probably want to buy a few blank cassette tapes, ready for recording your own programs. Most people prefer to have fairly short tapes, as this makes it quicker to find the programs on them. Special computer tapes are sold cheaply. The C10 and C15 lengths are just about right for most purposes. It is possible to use ordinary audio tapes with the computer but it is preferable to use tapes which have been prepared for computers. The reason is that small blemishes in the magnetic coating may have little effect when you are playing a recording of music or voices, but can have a disastrous effect when you are trying to save or load a computer program. They will not cause any actual harm to the computer, but the programs will not work properly, if at all. Therefore it is best to use only tapes which have been certified as suitable for computers.

Some manufacturers advertise tapes which are 'leaderless'. This means that they do not have the small length of coloured non-magnetic tape at either end. The advantage of this is that, having rewound the tape to the start, there is no risk of you trying to record

on the non-magnetic part. If you do this, you lose the beginning of your program and have to repeat the recording. But it is not too difficult to get into the habit of running the tape on a few inches to begin with, so it is not essential to have leaderless tapes.

Your friend VIC

The VIC is presented as a friendly computer. There are many computers which could never be regarded in that way, but what is so different about the VIC? For one thing, it has a robust keyboard with proper keys which have a firm feel to them. They are boldly marked so that you can see what they do. All these features give the user confidence in handling the machine. The layout of the keyboard has been designed to make it easy to type in programs, and there are special keys which can be used as control keys for games and other programs. These extra keys help keep things simple and easy to understand for the user.

One very good feature is the video display. Without any programming complications, the VIC produces bold, clear and colourful pictures on the TV screen. Any messages it displays are in large letters, which are easy to read. They can be in several colours, to make their meaning easier to understand. The screen can take any of sixteen different colours, with the border being any one of eight colours. These colours and the wide range of graphics symbols make it possible to design attractive and impressive displays, as we shall see in Chapter Three.

The sound chip has three 'voices', which means that it can play three different notes at the same time. The pitch of each note is controlled separately, so the melody may be accompanied by both treble and bass parts. The sound chip also has a 'white noise' generator which produces a range of noises from a rumbling like that of an aeroplane engine to a high-pitched hiss like that of escaping high-pressure steam.

By combining these sounds and notes in quick succession, which is very easy to do (see Chapter Five), many kinds of novel effects may be produced. The addition of sounds to a program makes it that much more effective and often more enjoyable or amusing. It is another aid to user-friendliness.

Well, the VIC and its other bits and pieces are lying unpacked on the table, waiting to be used. Let's move on to Chapter Two to find out how to put them together.

Chapter Two
Setting It Up

The VIC comes with all the cables necessary for connecting the system together. Before you start plugging in the cables, study the computer console itself to locate its main features. On top is the keyboard – much more about that later. At the upper right corner of the keyboard area is a red lamp. This lights when the VIC is switched on. The cable connectors are on the right-hand side and the rear side of the computer. On the right side (Fig. 2.1) we have a socket which takes the power supply lead from the power supply unit. This socket has '9V' marked beside it, as a reminder that a mains lead should *never* be plugged into it. Next to this is the switch for turning the VIC on and off. To the left of this is the control port. You can plug a variety of things into this, such as a games controller or a light pen. You can add these to your system later, if you want to.

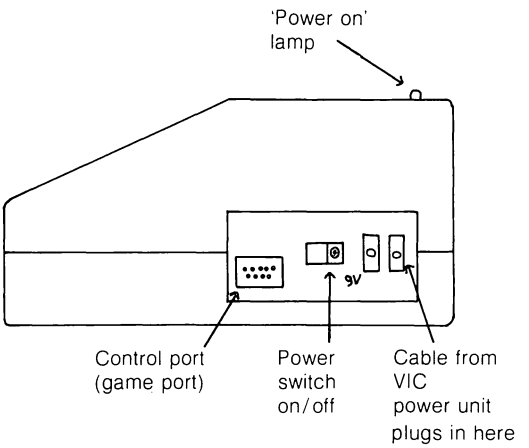


Fig. 2.1. The right-hand end of the VIC 20 keyboard.

The rear of the machine has five apertures (Fig. 2.2). On the left (as seen from the back of the computer) is the memory expansion port. This is where you plug in cartridges or additional memory

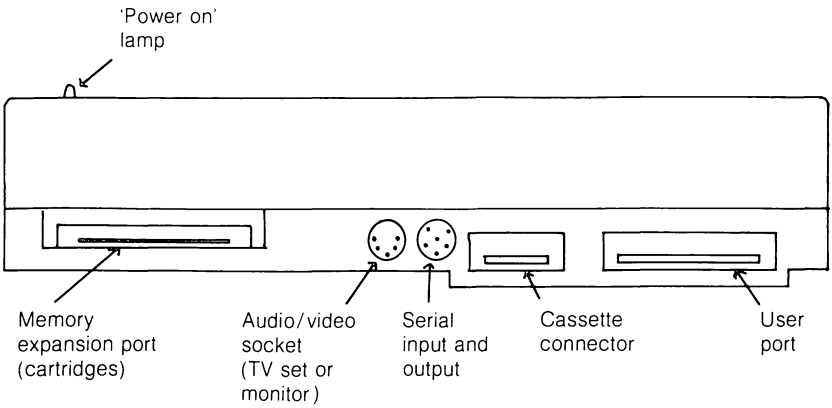


Fig. 2.2. The rear side of the VIC 20 keyboard.

boards. These can be very useful to have later, but you are not likely to need them at the beginning. Next come the two circular 'DIN' connector sockets. You will see that the pins of these are differently arranged, so it is impossible to plug a connector into the wrong socket. Just another example of user-friendliness! The audio/video socket is the one used for connecting the modulator box which sends the picture and sound signals on to the TV set. The other circular socket is for serial input and output. We will not have anything to do with this in this book. Briefly, it is where the computer sends messages to other devices. For example, if you plug a modem into this socket, the modem sends messages to other VICs through the ordinary telephone system. It can receive messages this way too. A printer with serial input can also be connected to this socket.

The fourth aperture on the rear face contains the cassette connector. This is a broad tongue-like edge of the computer circuit board. It has six copper strips on its upper surface which we call an edge-connector plug. The cassette recorder is connected here: its cable has a six-way edge connector socket on it. You will note that there is a notch (or keyway) cut in the edge connector between the second and third copper strips. The socket on the cassette cable has a key (Fig. 2.3) in the corresponding position. When the socket is pushed on to the plug, this key slides into the keyway. Having the key and keyway makes certain that you cannot connect the socket the wrong way round.

To the extreme right of the back surface is the user port. This is another place at which external devices such as a games controller or joystick, a light pen, a modem or a printer may be attached.

Having found your way around the computer case, place it on a table where you can sit comfortably to use it. You will need to have

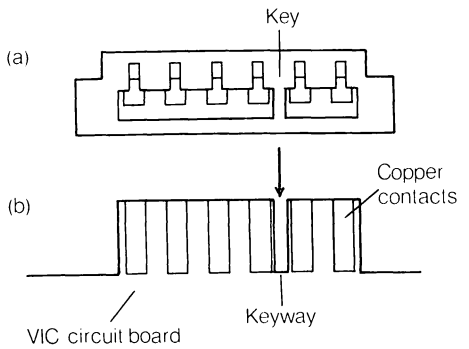


Fig. 2.3. Connecting the cassette recorder to the VIC 20. (a) End-on view of the socket on the recorder cable. (b) Top view of the circuit board, as seen through the cassette port opening.

two mains outlet sockets, reasonably close to the table. One is for the computer power supply. The other is for the TV set. The TV set is best placed behind the computer, though you can put it to one side of the computer, if the table is not deep enough from front to back.

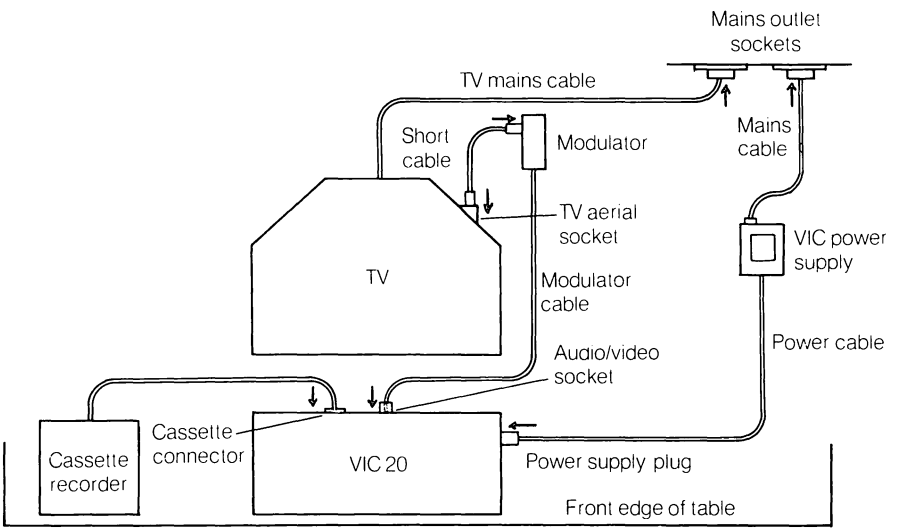


Fig. 2.4. Connecting up the system. The arrows indicate where a plug has to be pushed into a socket.

The next step is to connect the parts of the system together. The RF modulator has a long cable and a short cable (Fig. 2.4). The long cable plugs into the video port socket on the rear of the VIC. The short cable plugs into the aerial socket of the TV set. The cable from the cassette recorder plugs into the cassette port – position it so that the key slides smoothly into the keyway on the edge connector.

Actually we shall not be using the cassette recorder until the end of Chapter Three, so there is no real need to connect it yet.

The power supply has two cables. One of these has a connector which fits into the power socket on the right-hand side of the VIC (Fig. 2.1). The other may already be fitted with a mains plug. If it has no plug, you will have to fit one yourself or ask an experienced person to fit it for you. Figure 2.5 shows the way of wiring a standard 13-amp mains plug. Note the colours of the three wires and make certain that they go to the correct pins.

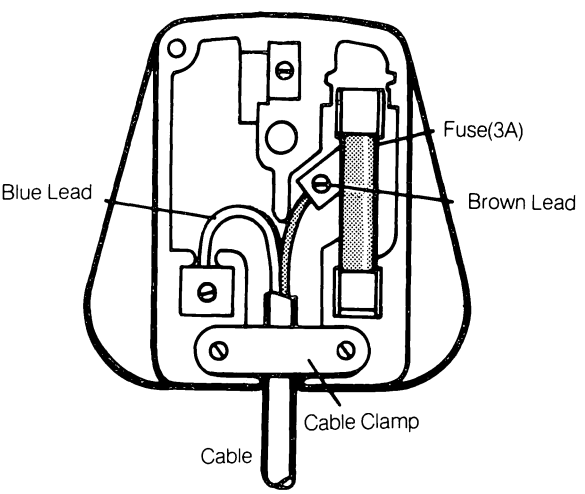


Fig. 2.5. Connecting a 13-amp plug to the mains cable.

Plug in the power supply and the TV set. The power supply has no switch so remember to pull the plug out (or turn off the outlet switch) when you have finished computing. Switch on the computer using the small switch on the right-hand side of the case (Fig. 2.1). The red lamp on top of the computer should now light.

Switch on the TV set. Probably nothing will happen, because the set is not yet tuned to the computer. The VIC's modulator is set to send signals to the TV set on channel 36. If yours is a push-button set, choose one of the buttons which you do not use for regular TV programs. Push the button and adjust the corresponding tuning knob until you obtain a sharp picture, in colour, looking like Fig. 2.6. Different makes of TV set differ in the way the channels are selected and tuned, so consult the instructions for your set, if in doubt. It is usually only a matter of a few tens of seconds before the VIC power-up display is shining brightly on your screen. The central

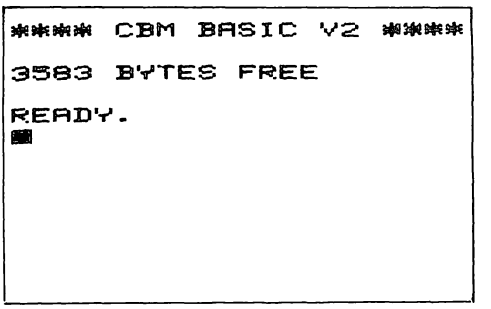


Fig. 2.6. The screen display when the VIC 20 is first switched on.

area of the screen is white with blue letters. It is surrounded by a border of paler blue.

Adjust the tuning carefully to eliminate double-images and colour fringes as far as possible. If you are able to reserve that push-button for future use by VIC, the set will be ready to use whenever it is switched on.

Finding your way around the keyboard

If you have used a typewriter or another computer, you will probably know where to find most of the keys. If not, it is surprising how quickly you get used to it. It is said that the odd arrangement of keys (the QWERTY keyboard, as it is usually called) was first designed for the original typewriters to make it hard for typists to type quickly! The early mechanisms jammed too readily when used by a high-speed typist. The VIC responds to key presses in a fraction of a millisecond, so there is no chance of jamming its electronics, but we are stuck with what has now become the standard keyboard layout. One thing you need not worry about is pressing the wrong key at the wrong time. Press any key you like whenever you like – it will not harm the VIC. Of course, it may do odd things to the picture on the screen and to any program which happens to be in memory, but that is an entirely different matter. You can always recover from such a situation and begin again.

To begin with, try pressing any of the keys. Try typing your own name, or the name of your house, or just anything else that takes your fancy. If you want to make a space, press the 'space-bar', the long bar at the front of the keyboard. Whatever you type appears on the screen in large capital letters – perhaps mixed up with a few numbers and punctuation marks, if you have pressed wrong keys by mistake!

If you *do* make a mistake, there is an easy way to correct it. Just press the key at the top right corner of the keyboard. It is marked:

The 'DEL' is short for 'delete', which is just what it does. The computer back-spaces one space, and the wrong letter you have just typed disappears from the screen. You may already have made another discovery about this key. If you press it and hold it down for a second or so, it carries on and deletes many more of the letters to the left.

All the time you have been using the keyboard you will have noticed the blue square which 'blinks' regularly on and off. This is the *cursor*. It is a sort of marker showing you where on the screen the next letter you type will appear. As you type, the cursor moves across the screen, staying one space ahead of the most recent letter. At the end of the line it flips down into the next line below, to the first space on the left. When you press the 'delete' key, it moves to the left, cancelling the letters as it reaches them.

It is surprising what you can do with the cursor! Try pressing the key marked:

This key is next to the INST/DEL key. Pressing this key sends the cursor to its 'home', which is the top left-hand corner of the screen. Incidentally, when we use the word *screen*, we do not mean the whole TV screen, but the central area (now white) on which the letters appear. The coloured area around it (now light blue) is called the *border*.

Now we will use two keys at the bottom right-hand corner of the keyboard to run the cursor all around the screen. These keys are both marked CRSR, which is short for 'cursor'. The arrows show which directions the cursor is made to move by pressing these keys. Pressing the up/down key makes the cursor move down, line by line. How can we make it move upward, line by line? The way to do this is to press a SHIFT key. What happens when you press the shift key at the same time as the left/right CRSR key?

There are two keys called SHIFT on the bottom row. We need two keys because we use SHIFT rather a lot. We never use SHIFT

alone but always press it at the same time as some other key. Having two keys makes it easier to press one key with a finger of one hand, while holding down one or other of the SHIFT keys with a finger of the other hand. The keyboard is very user-friendly to two-fingered typists! Try the CRSR keys both with and without SHIFT and you will soon learn to steer the cursor where you want it to go.

By this time the screen will probably be covered with a real mess of words, letters and figures. Here is another use for the SHIFT keys. Press SHIFT at the same time as you press the CLR/HOME key. Now we see what CLR stands for. It means 'clear'; not only does the cursor get sent home, but everything else gets cleared from the screen. You are left with a clear white area ready to start again.

This time, try pressing SHIFT at the same time as you press other keys. Some of the keys have two punctuation marks on them, for example:

Pressing the key unshifted (that is, without pressing SHIFT at the same time) gives you the lower mark (/). Pressing SHIFT and the key at the same time gives the upper mark (?). This is just the same as on an ordinary typewriter. In the same way, the keys with a figure and a punctuation mark or symbol (for example):

give the figure (4) when unshifted and the symbol ($) when shifted.

You can now understand why the CLR/HOME key just sends the cursor home when pressed alone, but clears the screen too when pressed with the SHIFT key. There is one small exception to this. Pressing the 'up-arrow' (↑) key alone prints the 'up-arrow' on the screen. Pressing the 'up-arrow' key and SHIFT together, gives the Greek letter 'pi' (π). This is marked on the *front* of the 'up-arrow' key, not on the *top*, above the 'up-arrow', as you might have expected.

Have you tried shifting the letter keys? If not, try some now. Letter 'A', for example, gives capital 'A' when pressed alone. When you press it with 'SHIFT' you get something quite new – a 'spade' symbol, like that used on playing cards. You may have noticed that this symbol is on the front of the 'A' key. In fact there are two

symbols on the front of every letter key, as well as on the fronts of the '+', '−' and '£' keys in the top row and the '@' and '*' keys in the next row down. The symbols you get are the right-hand one of each pair. We call these symbols the *right side set*. We will see how to get the left side set in a moment.

Help!

It could be by this stage that you are becoming a little mystified. Maybe things are happening on the screen which do not *quite* agree with what has been said above. This is probably because you have pressed one or more of the other control keys at the wrong moment, either by mistake or perhaps by way of experiment. You may even have seen the sinister words 'SYNTAX ERROR' appear on the screen (not quite as friendly as usual!).

Do not worry! There is an easy way of getting things back to normal. Find the RUN/STOP key, which is on the extreme left, and the RESTORE key, which is on the extreme right. Press them both together. All the odd letters, symbols and threatening messages are cleared away. The encouraging word READY appears at the top left corner of the screen, with the cursor blinking away peacefully below it. The VIC is now restored to the state it was in when first switched on. If VIC had taken to writing letters in red or some other colour, it will now be writing in its familiar blue again.

It sometimes hapens that *nothing* appears on the screen when you press the keys. This is most likely to be because you have altered the colour of the writing to white. Writing in white ink on white paper is not exactly the clearest way to make things clear! As explained above, pressing RUN/STOP with RESTORE brings back the blue again.

Colours

It is not a bad idea to play around with the keys for a few moments longer. Try all the right side symbols to see what they look like on the screen. Maybe you can key in blocks of them, building them up into pictures. Figure 2.7 shows a few examples. But it might be nice to get away from boring old blue. The VIC has so many colours that it is a pity to stay with one. You will already have spotted the names of the colours on the fronts of the number keys. To change colour you use

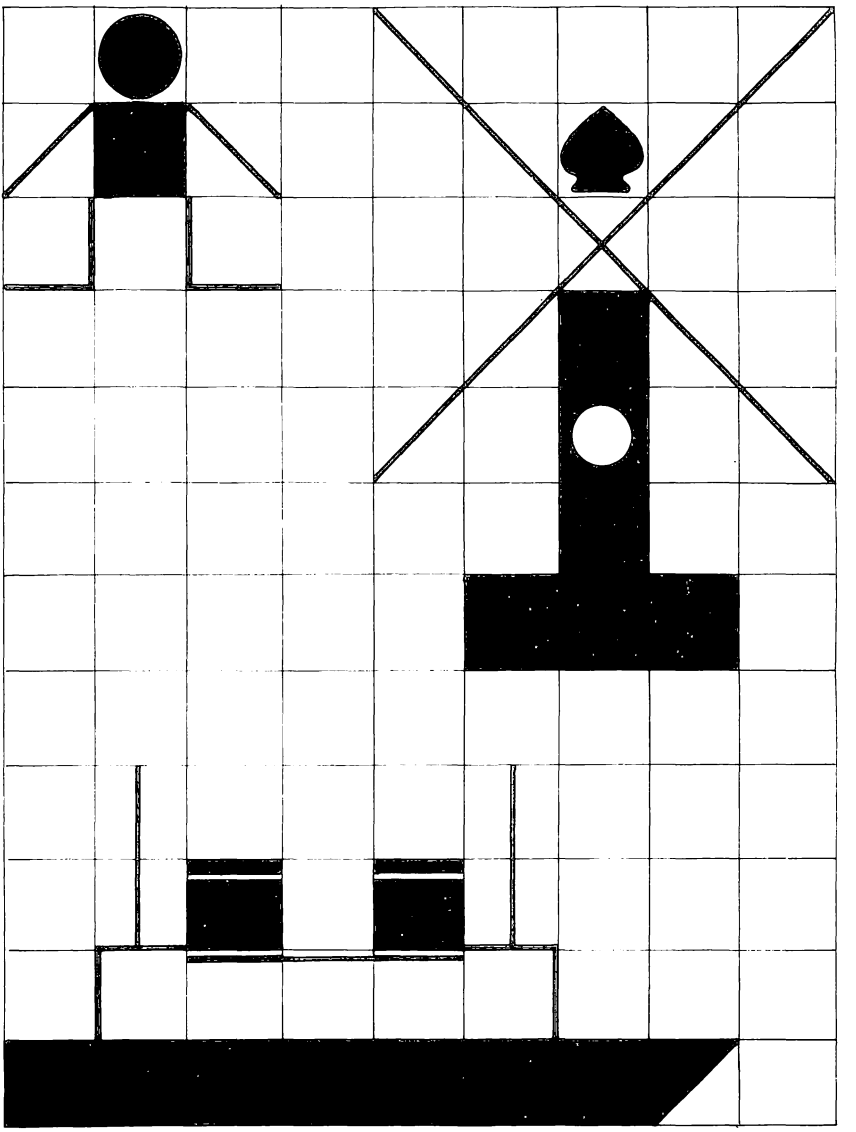


Fig. 2.7. Pictures made up from right-side graphics.

the CTRL (control) key, which is on the extreme left. Use it like a SHIFT key, with one of the number keys. Try pressing CTRL and the '3' key. This has RED marked on its front so, as we might expect, all future letters and symbols on the screen are in red. The cursor changes colour too, so you will always know what colour we are using.

Now we can see why letters and the cursor can sometimes

disappear. Press CTRL and key '2', then type a few letters. The cursor, now white, is invisible against the white screen, and so are the letters you have typed. They are really there, but invisible! Press CTRL and key '6'. The cursor reappears some way down the screen, showing that it did move when you typed in white. Now it is green.

Number keys '9' and '0' do not have colours marked on their fronts, but have the shortened words RVS ON and RVS OFF instead. To find out what these mean, press CTRL and key '9' (RVS ON) together. Now type a few letters or right side symbols. You get white letters or symbols on a green background. For white on red, press CTRL with key '3'. Obviously, RVS ON is short for 'reverse graphics on'. Try other reverse colours. We shall use reversed letters later, to make headings and other important words stand out boldly on the screen. Reversing the symbols doubles the range of designs available for creating interesting pictures. When you want to get back to normal, unreversed letters and symbols, press CTRL with RVS OFF. Or, if you like, you can restore things by pressing RUN/STOP and RESTORE.

Left side, right side

There is an oddly marked key at the extreme left of the front row. It shows a 'C' partly surrounding a little flag. It is the Commodore trade mark. We will call this the 'Flag' key. Try using 'Flag' as a kind of 'shift' key. At long last, you get the left side set of symbols. For example:

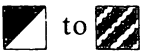
Pressing key 'A' alone (unshifted) gives 'A'
Pressing key 'A' with 'SHIFT' gives a ' '
Pressing key 'A' with 'Flag' gives a ' ' symbol

Put some of the left side set on the screen to see what they look like. Try them in several colours, and also reversed. Type a few right side symbols as well. Aim to cover most of the screen with a mixture of letters, numbers, right side symbols and left side symbols. Now we are ready for the VIC 'All Change' act (or almost 'All Change', anyway!).

The 'All Change' effect occurs when you do something that you might have already wondered about: what happens when you SHIFT the 'Flag' key? Perhaps you have already tried this – some people have very itchy fingers! If you have not, try it now – press 'Flag' and SHIFT together, and watch what happens on the screen.

A lot happens at once. To put everything back as it was, press 'Flag' and SHIFT again. Then press them again to repeat the 'All Change' act.

The effects of the 'All Change' are:

	Example
Capital letters change to 'small' letters	A to a
Right side set change to capital letters	to A
Some of the right side set change to different symbols	

The numbers, punctuation marks, mathematical symbols (+, −, =, etc.) do *not* change. Neither do most of the right side set. Colours do not change either.

All this seems rather confusing at first but the idea behind it is simple. The VIC has two ways of putting things on the screen. They are called modes. One is the *Graphics Mode* and the other is the *Text Mode*. You change from one mode to the other by pressing 'Flag' and SHIFT together.

When the VIC is first switched on (or when you have restored it to its switch-on state by pressing RUN/STOP with RESTORE), it goes straight into Graphics Mode. This gives you capital letters and the right side set (using SHIFT). You also get numbers and mathematical symbols. The Graphics Mode is best to use when you are keying in programs.

If you put the VIC into Text Mode, by pressing 'Flag' with SHIFT, you get 'small' letters (lower-case) when you press letter keys without shifting. To get capital letters you now have to press the SHIFT key, just as on an ordinary typewriter. This mode is best if you are wanting to use the VIC like a typewriter. You may be wanting to key in a lot of text, such as the instructions for playing a game, or you may be using the VIC with a word processing program. Text Mode also gives you the left side set (pressing 'Flag' on its own) which has most of the symbols useful for preparing charts and tables.

The blackboard

This will give you plenty of practice in using the control keys, as well as giving full rein to your artistic talents. First of all, type:

```
POKE 36879,15
```

Then press the RETURN key. The screen goes black, and the border

becomes yellow. The black screen is the 'blackboard'. Now press 'CTRL' and '8' together. The cursor changes from white to yellow. This is the 'chalk'. Next press CTRL with 9. This puts the VIC into 'reverse on' mode, so that a 'space' is printed on the screen as a yellow block. To clean the blackboard, press SHIFT with CLR/HOME.

The cursor goes to its home at the top left corner of the blackboard, which is cleared of everything you have already typed. You steer the cursor around the screen by using one of the two CRSR keys to move it down or to the right. Pressing SHIFT at the same time as a CRSR key moves the cursor up or to the left.

When you want to draw a line with a chalk, position the cursor to the left of where the line is to be, then press the space-bar. Press once to draw a single yellow block. Hold it down to draw a horizontal line several blocks long. Then move the cursor along to where you want the next line to begin and press the space-bar again. Vertical lines are drawn block by block, moving the cursor down and to the left between each block.

Change the colour of the chalk whenever you like, by pressing CTRL with one of the number keys 1 to 8. To delete any block you have drawn, position the cursor to the right of the block, and press INST/DEL. Then press SHIFT with INST/DEL. You can start afresh at any time by pressing SHIFT with CLR/HOME. Happy scribbling!

Keys to the keys

All this takes a little getting used to, but after a while becomes almost second nature. There is no point at all in your trying to learn at this stage what every key does. As you use the VIC, you will soon get to know what the most frequently used keys do. As you become more familiar with the machine, you will get to know what the less frequently used keys do too.

The first four reference tables in Appendix A are there to help you whenever you get stuck. Tables 1 to 3 list the keys and what they do. Table 4 works the other way round. When you want to do something but do not remember how to do it, look in this table to find out which keys to press. You will probably use these tables a lot at first, but less and less as you get used to the VIC.

Chapter Three
A Splash of Colour

If you cannot wait for the colours to start flashing on your screen, jump straight ahead to the program in Listing 3.3. But if you follow this chapter, step by step from the beginning, you will soon learn a lot about getting the VIC to do its colourful best.

Switch on the VIC. Then press CLEAR/HOME with SHIFT, to clear the screen and send the cursor to its home at the top left corner. In order to get the best effect from the colours, it is a good idea to provide a suitable background. One of the easiest things to do on the VIC is to change the colour of the screen and the border which surrounds it. One of the commands to do this is:

```
POKE 36879, 2Ø7
```

Try it – type it in just as above, taking care to get the numbers exactly right. As you type, the command appears on the screen, but nothing else happens. The VIC is simply taking what you type and 'printing' it on the TV screen. Now press the RETURN key. The effect is instantaneous. The colour of the screen changes from white to light purple, while the colour of the border changes from pale blue to yellow.

The command 'POKE' is one which is used a lot, so let us see what it means. As stated in Chapter One, the VIC has a memory which you can *change*, called RAM. RAM is short for 'Random Access Memory'. The main point about RAM is that any information you store in it stays there until you alter what you have stored, or until the VIC is switched off. RAM actually consists of a set of electronic circuits but think of it as a set of little boxes each with a slit in its lid (Fig. 3.1). Each box has a number, to identify it. We call this number its *address*. The address of the box you just used is 36879. This is a special box (or unit of memory) set aside for receiving instructions about what colours the screen and border are to be. When you type 'POKE 36879,2Ø7' it is as if you were to write the number '2Ø7' on a

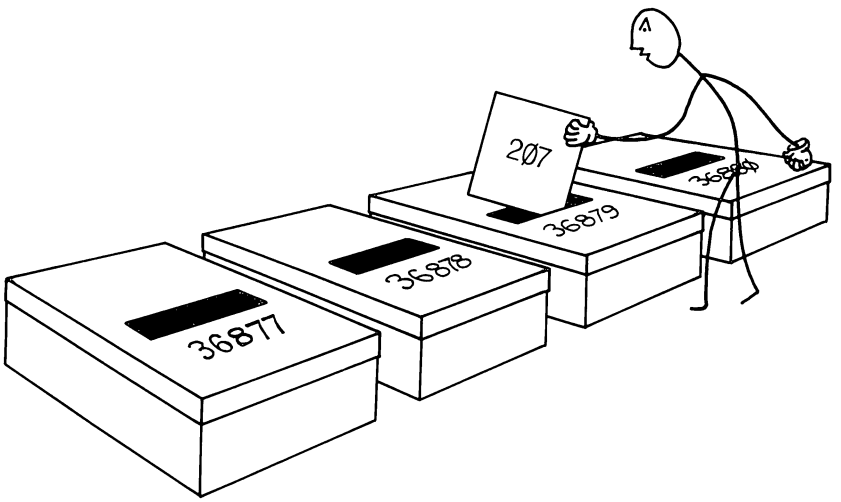


Fig. 3.1. POKEing the value 2Ø7 into RAM at address 36879.

piece of paper and poke it through the slot in the lid of the box. When you are POKEing numbers into RAM do not worry that you might POKE numbers into ROM by mistake. ROM is quite unchangeable and POKEing has no effect on it.

When you type 'POKE 36879,2Ø7' the VIC prints your command on to the screen, but does not do anything about it. But, when you press RETURN, you are saying to the VIC, 'There, *that's* what I want you to do. Go and do it!'. The VIC then does exactly as it is told. It takes the number '2Ø7' and POKEs it into address 36879 in its memory. As soon as that number is POKEd in, the part of VIC concerned with the video display finds this new number at address 36879 and changes the screen and border colours as instructed.

Numbers that are POKEd into an address may have any value from Ø to 255, though often the number needs to be within a more limited range (say 1 to 8) if it is to have any effect. This depends on which address is being POKEd. Table 5 (see Appendix A) shows which numbers are allowed at address 36879, and what effect they have on screen and border colours. Try some of these now. Type in 'POKE 36879,', followed by any of the numbers from Table 5. Then press RETURN. Do not forget the comma between the two numbers. If you leave this out, VIC does not understand what you mean, and replies with '? ILLEGAL QUANTITY ERROR'. VIC then prints 'READY' to let you know that you should try again. Incidentally, if you make mistakes when typing these numbers, as everyone does at times, use the INST/DEL key to back-space and

delete the wrong figures, as described in Chapter Two. Then type in the correct figure.

Now to start drawing (or printing, as it is more usually called) some shapes on the screen. Type the same command to get back to the light purple screen with yellow border. Clear the screen and send the cursor back home. Now type this command:

```
POKE 7777,9Ø
```

Press RETURN after typing this. You may not notice that anything happens, but there is now a white diamond shape on the screen.

As before, you have typed in the word 'POKE' followed by two numbers. As before, the first of the numbers is an address in RAM. The VIC has a part of its memory set aside to hold information about what is supposed to be on the screen. There can be 23 rows of characters on the screen and each row can hold 22 characters, so that altogether it is possible to put up to $22 \times 23 = 506$ characters on the screen. The screen RAM needs 506 addresses, one for each of these characters. This is the *Character Code RAM*. As Fig. 3.2 shows, the address corresponding to the top left character is 768Ø. The bottom

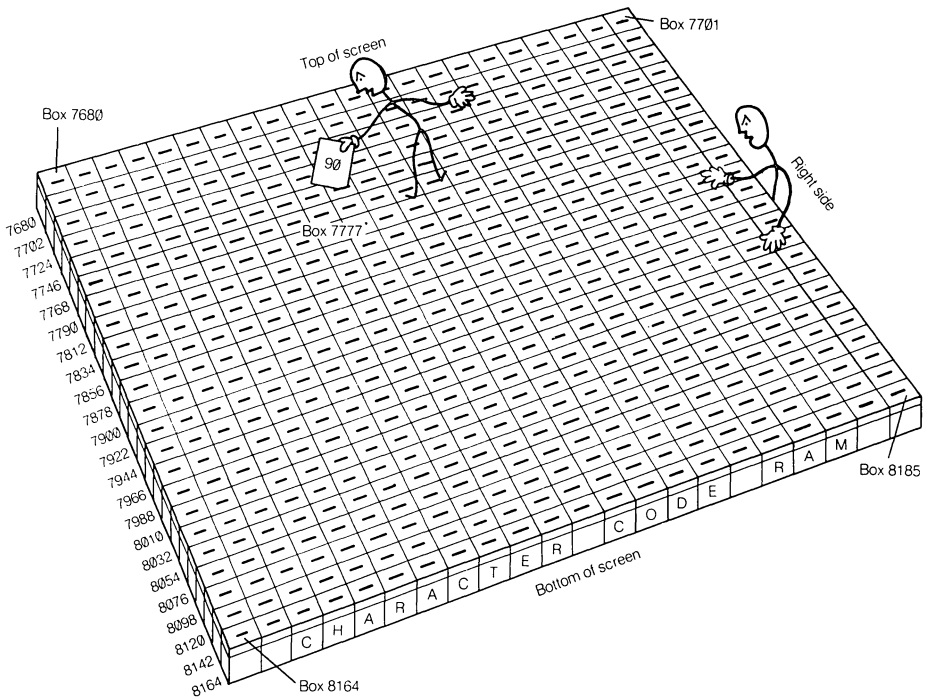


Fig. 3.2. POKEing the code for a 'diamond' into the character code RAM at address 7777. (*Note*: The addresses are different if the VIC has more than 3K of extra RAM added; see Appendix B.)

right corner is address 8185. The diagram shows that the white diamond appeared where expected when you POKEd address 7777. Try POKEing some other addresses, for example:

```
POKE 7778, 81
```

Press 'RETURN' and see the white disc appear on the screen immediately to the right of the diamond. It is clear that the *second* number you POKE tells the VIC what shape or other character it is to print. This is the *Character Code*. Table 6 shows you which numbers to use to get any symbols or letters you want. Try POKEing a few of these codes to get the feel of the system.

Don't scroll it!

When you type on the bottom line of the screen and press RETURN, everything on the screen moves up one line. This is called *scrolling*. Normally this does not matter, but if you have POKEd a 'diamond' into 7777, for example, the computer takes the 'diamond' out of 7777 and puts it into 7755, so that it is displayed one line further up the screen. The colour code for the 'diamond' is also moved one line up (from 7777 + 3ϕ72ϕ to 7755 + 3ϕ72ϕ). If you want to alter the colour of the diamond *now*, it will be no use POKEing 7777+3ϕ72ϕ. After the screen has been scrolled a few times, you will probably have lost track of where everything is.

To avoid scrolling, never let the cursor get to the bottom line. Before you start each sequence of commands, clear the screen and send the cursor home (CLR/HOME with SHIFT). This usually gives you enough lines in which to do anything you want.

If you are getting tired of typing 'POKE' every time, use the short version of this command. The VIC understands short versions of all its commands or keywords (see Appendix D of *Personal Computing on the VIC 20*). It is worth memorising those you use often. To type the short form of POKE, type 'P' then type 'SHIFT' and 'O' together. On the screen you will see 'P' followed by the right side symbol of the 'O' key.

If you still have the diamond and disc in position, you are ready to

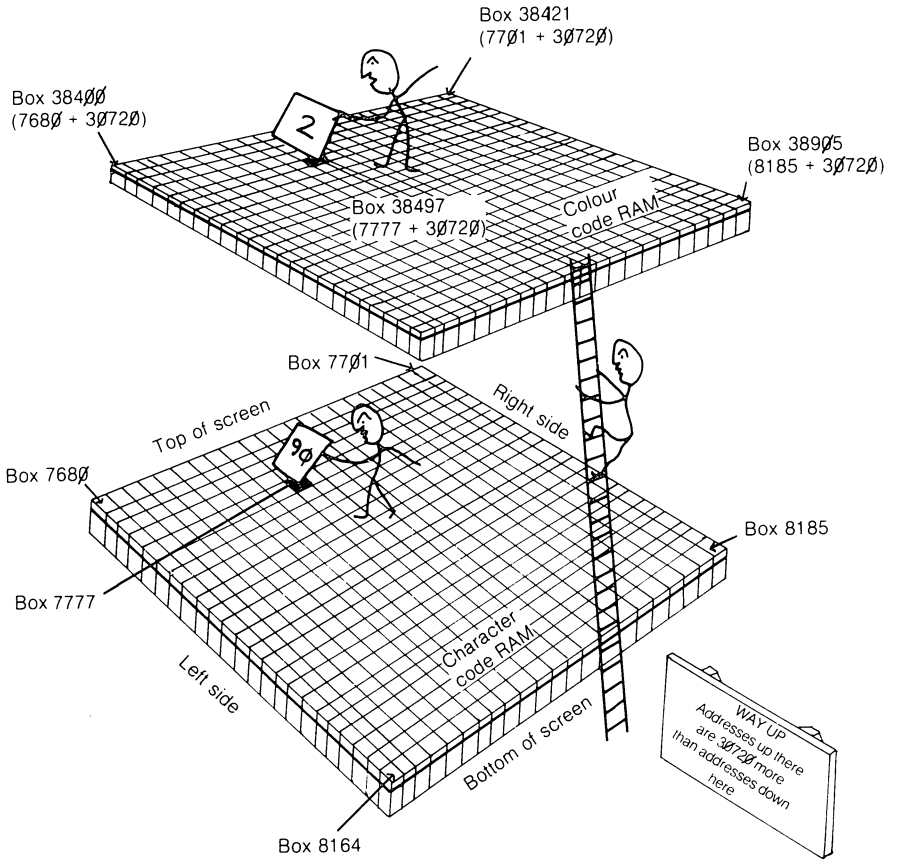


Fig. 3.3. POKEing the colour code for 'red' into colour code RAM address 38497, to give a red colour to the 'diamond' POKEd into character code RAM. (*Note*: The addresses are different if the VIC has more than 3K of extra RAM added; see Appendix B.)

carry on. If not, clear the screen and repeat the two POKEs already given. The diamond and disc are white because you have not yet told the VIC to make them any other colour. You can tell the VIC what colour you want by POKEing a number into another address. The VIC has a second set of 506 addresses which hold the colour instructions. The addresses of the *Colour Code RAM* run from 384ØØ to 389Ø5 (see Fig. 3.3). The sequence of addresses in this RAM exactly matches the sequence in the Character Code RAM. Each address in Character Code RAM is matched by an address in Colour Code RAM. The difference between the two addresses is always exactly 30720. To change the colour of the diamond, we first calculate its Colour Code address; 7777 + 30720 = 38497. Now type:

```
POKE 38497, 2
```

The diamond changes to red. (Did you remember to press RETURN? You *always* need to do this to get the VIC to take action). The '2' is the colour code for 'red'. Table 7 in Appendix A lists the eight colour codes. Now try turning the white disc into a green one. First calculate 7778+30720 = 38498. The Colour Code for green is '5'. Then type:

```
POKE 38498,5
```

and the disc becomes green. Experiment a little with changing the colours of these, and other characters, on the screen. Try putting three discs (Code 81) on the screen one below the other, with the top one at address 7787. There are 22 characters on a line so the addresses of the other two discs are 7787+22 = 7809, and 7809+22 = 7831. Then make the discs red, yellow and green, like a set of traffic lights.

Are you getting tired of adding up these long numbers? Why not let the VIC do it for you? Instead of asking the VIC to 'POKE 78ø9,81' just type:

```
POKE 7787 + 22,81
```

The VIC does the addition *before* carrying out the instructions. In the same way you can change the colour of the disc at 7787 + 22, by getting the VIC to add the 30720 as well:

```
POKE 7787+22+30720,7
```

This gives you more typing to do, but there is less arithmetic.

Shifted '+'

On many other computers you have to press the 'SHIFT' key in order to type '+'. This may become a habit which is hard to break. If you press SHIFT with '+' on the VIC, you get a graphics sign that looks very much like a plus sign. On reading what you have typed, you may not notice your mistake. But the computer does not accept this and various kinds of error can arise later. So when you want the micro to add, type '+' *without* the SHIFT key.

A colour program

The set of traffic lights can be drawn by typing in six commands and pressing RETURN after each. White discs appear, one by one, and are then given colours, one by one. Instead of getting the micro to carry out your instructions one by one, you can give it a whole list of instructions and then tell it to carry out the lot in quick succession. The VIC will still do these things one at a time, but works so fast that it appears *to you* as if they were all done at once. When you give a micro a list of several instructions, you are giving it a *program*. A traffic-light program is shown in Listing 3.1.

```
10 POKE 7787,81
20 POKE 7787+22,81
30 POKE 7787+44,81
40 POKE 7787+30720,2
50 POKE 7787+22+30720,7
60 POKE 7787+44+30720,5
```

Listing 3.1. A traffic-light program.

The VIC has been left to do the addition! A program has a number at the beginning of each line. This is the *line number*. Type in the program. Press RETURN as you finish typing each line. The computer does not act on each instruction straight away. The line number at the beginning of each line tells it that this is a program line. It is to be remembered (stored away in memory) but *not acted on* until later.

If you make typing errors, it is best to press RETURN and then retype the line again (not forgetting the line number). The VIC then forgets the old line which had that number, and remembers only the new line. When you have finished typing in all six lines, check again for mistakes. Retype the line if you find any. It does not matter, if you have to retype in one of the earlier lines after you have finished typing the later lines. The lines are always arranged in order of line number. If you have had to retype some lines, and want to see the final result in proper order, just type LIST (followed by RETURN) and the complete program is displayed with all its lines in order.

Now you have a correct program in the VIC's memory. To get it to act on the program, type RUN. Still nothing happens, but when you press RETURN, it does as it is told and RUNs your program. In an instant the three discs appear on the screen in their correct colours. It all happens so quickly that you do not have time to see that they are white at first and change colour afterwards.

To see this happen again, clear the screen (SHIFT and CLR/HOME), then type RUN, followed by pressing RETURN. Run the program as often as you want. If you like, you can alter it to

obtain different colours, or to draw different characters. Retype any of the lines, to replace an old line by a new one.

The Magic Carpet

The Magic Carpet is a program which is short, easy to understand, but produces an exciting splash of colour. The best way of understanding how it works is to build it up step by step. This means typing in the lines out of order but, as explained earlier, this does not matter to the VIC. If you have come straight from the previous section, the old program is still in memory and the screen is probably covered with words and symbols. To get ready for the new program first type NEW and press RETURN (this gets rid of the old program) and then press RUN/STOP and RESTORE to restore the screen to its switch-on state. Begin by typing lines 5Ø to 8Ø (see Listing 3.2).

```
50 FOR J = 7680 TO 7701
60 POKE J,81
70 POKE J+30720,2
80 NEXT J
```

Listing 3.2. The first step toward the Magic Carpet program.

Run the program (type RUN, then press RETURN). A row of red discs appears across the top of the screen. The middle two lines are easy to understand. Line 6Ø is POKEing a disc-shaped character (81) to an address numbered 'J'. J can take different values as will be explained later. Line 7Ø uses an address which is bigger than J by 3Ø72Ø. If J is an address in the Character Code RAM, then J + 3Ø72Ø is the corresponding address in the Colour Code RAM. The number '2' makes the discs red. The interesting lines are lines 5Ø and 8Ø. Line 5Ø tells the VIC to do something (the two POKEs) several times. The first time it does it, the value of J is to be 768Ø. This is the address of the top left of screen (see Fig. 3.2). At line 6Ø a disc is printed there and at line 7Ø the disc is made red. At line 8Ø, the command 'NEXT J' causes J to be made bigger by 1. J is now 7681. The VIC goes back to line 5Ø and *repeats* its action, only this time J is 7681 and the disc is printed at the second place on the top row. This process is repeated with J taking the values 7682, 7683, 7684, and so on, until J reaches the value 77Ø1 (line 5Ø says 'FOR J = 768Ø TO 77Ø1'). This is the address of the right-most position on the top row. At this point VIC has printed a full row of discs. J has now reached its maximum value, so the program stops and prints 'READY' on the next line but one.

It would be good to be able to print discs of any colour, chosen by chance, so that we do not know which colour will be printed each time we run the program. To do this we need a program line such as:

```
7Ø POKE J + 3Ø72Ø, C
```

in which C can have any value from Ø to 7, to give any of the colours listed in Table 7 in Appendix A. The way to do this is to use a special BASIC function called 'RND'. This is short for 'random' which means 'by chance'. If we use the statement 'C = RND(1)', the VIC gives C a value between Ø and (almost) 1. Try this and see – it is not part of the program, but just type:

```
PRINT RND(1)
```

and press RETURN. A ten-figure number is printed, preceded by a decimal point. You get a different number every time you do this and there is no way of telling which number you will get. Since the numbers range from 0.0000000000 to 0.9999999999, it is likely to be a long time before you get the same number again! This number is no use as a Colour Code because Colour Codes must be *whole* numbers (called *integers*) between Ø and 7. Try this:

```
PRINT RND(1)*8
```

Now you get numbers in the range Ø to (almost) 8. The part after the decimal point is chopped off and 'forgotten' by using another function, called 'INT' (short for 'integer'), like this:

```
PRINT INT(RND(1)*8)
```

Take care with the brackets! It is a general rule that, in any expression, the number of brackets facing one way must equal the number facing the other way. Try this command several times. It gives random values which are integers (whole numbers) between 0 and 7. The symbol '*' means 'multiply'. The random number between 0 and 0.9999... is multiplied by 8 to give a number between 0 and 7.9999... You never get '8' because the largest possible random number is just less than 1.

Now put this into the program. Type a new line:

```
2Ø C = INT(RND(1)*8)
```

and alter line 7Ø to

```
7Ø POKE J + 3Ø72Ø, C
```

Run the program several times. Each time you run the program the

value of C and, hence, the colour of the discs is picked at random. Sometimes it may make C = ∅ and you get a row of white dots which are invisible against the white screen. You can always make them visible by POKEing 36879 (see Table 5 in Appendix A). If the RUNs and READYs get near the bottom of the screen, clear the screen and start again.

Lost? - then LIST

When you are typing in programs, especially when you are making corrections, adding new lines or retyping old ones, it is easy to forget exactly what you have typed. You can see the whole of your program at any time by simply typing LIST, then pressing RETURN. Do this often - it helps you to get a clear idea of the current state of your program.

Now to enhance the program further. As well as having random colours, let's have the lines printed randomly across the screen. Instead of printing along the top row only, let the VIC select a row at random. Since there are 23 rows, numbers 0 to 22, VIC needs a random number between 0 and 22. We can use the same kind of command as before. This time it is:

```
X = INT(RND(1)*23)
```

The expression gives random integers between 0 and 22, one for each row. The addresses of the left-hand position in each row can be seen in Fig. 3.2. The first address of each row is found by taking 768∅ and adding 22 as we go down the screen. To find the first address of row X, 768∅ + X*22 must be calculated. To find the last address in that row, 77∅1 + X*22 must be calculated. Type this line which chooses a value for X:

```
3∅ X = INT(RND(1)*23)
```

Then change line 5∅ to:

```
5∅ FOR J = 768∅ + X*22 TO 77∅1 + X*22
```

Run the program several times. Now you get lines across the screen at different levels and in different colours.

It is very tedious to have to keep on typing RUN and it spoils the

picture to have all those RUNs and READYs scattered all over the screen. To get the VIC to repeat the program indefinitely, type this line:

```
15Ø GOTO 2Ø
```

GOTO is usually typed as one word, although you can also type it as two. Now run the program. When the program gets to the end, it is sent back to the beginning again. The program runs on and on. Gradually your typing is wiped out by rows of coloured discs. When you feel it is time to stop, press the RUN/STOP key.

What next? What about changing the screen colour itself and also the border colour. Random numbers seem an obvious way of doing this too. Table 5 shows that we need random numbers between 8 and 255. The expression:

```
INT(RND(1)*248)
```

gives numbers between 0 and 247, so add 8 to this to get numbers between 8 and 255. Type this new line:

```
4Ø POKE 36879,8+INT(RND(1)*248)
```

The address 36879 is the one we mentioned earlier in the chapter. Run the program. Now the colour really is beginning to splash!

One feature that spoils the effect is that anything which is on the screen to begin with stays there until it becomes covered by a row of discs. It would be better if the computer cleared the screen before starting the display. The way to do this is one that users of other machines may find strange. Instructions to the VIC are given in the form of a message. We can think of sending the VIC a message which is not printed on the screen during the program. Type in line 1Ø of the program, beginning:

```
1Ø PRINT"
```

Now press SHIFT and CLR/HOME together, just as if you were clearing the screen. The screen does not clear, but a reversed 'heart' symbol appears after the quote marks. Type the closing quote marks. Line 1Ø now looks like this:

```
1Ø PRINT"♥"
```

You have included a *Control Code* in the message. This code is never printed on the screen during the running of the program, but is read and interpreted as a command 'clear the screen'. Run the program

and see this happen. There are 23552 different patterns possible, so displaying them at a rate of about 160 a minute, two and a half hours would be needed to show them all!

Here is the complete program which not only prints *rows* of discs but alternately prints rows and *columns*. The second part (lines 9Ø to 14Ø) is very similar to the first. Do not worry about the 'STEP 22' in line 11Ø – this will be explained in Chapter Six. Type in lines 9Ø to 14Ø and run the program shown in Listing 3.3.

```
10 PRINT"□"
20 C = INT(RND(1) * 8)
30 X = INT(RND(1) * 23)
40 POKE 36879,8 + INT(RND(1) * 248)
50 FOR J = 7680+X*22 TO 7701+X*22
60 POKE J,81
70 POKE J + 30720,C
80 NEXT J
90 C = INT(RND(1) * 8)
100 Y = INT(RND(1) * 22)
110 FORJ = 7680+Y TO 8164+Y STEP 22
120 POKE J,81
130 POKEJ + 30720,C
140 NEXTJ
150 GOTO 20
```

Listing 3.3. The Magic Carpet.

Oh yes! Some readers may object that magic carpets are usually Eastern in origin. The colours and design of this one are too chunky and bold to have come from the lands of Aladdin. No problem! – while the program is running, press SHIFT and the 'Flag' keys together. This puts the VIC into Text Mode. The program runs just as well, except that instead of discs you get 'Q's. The delicate effect is the nearest you can ever get to a Persian carpet design.

To summarise

In this chapter, you have found out how to:

- change the colour of the screen and border
- make the VIC print any character (letter, number or symbol) anywhere on the screen
- make the VIC print the character in any of eight colours
- type in and RUN a program
- use the short form for BASIC keywords
- make the VIC do the adding
- generate random numbers in any range you want
- send control commands to the VIC in PRINT statements

You have learned that:

- pressing RETURN makes the VIC go into action
- pressing RUN/STOP stops the program
- PRINT tells the VIC to display characters on the screen
- POKE X, Y, puts value Y into address X of memory
- NEW gets rid of any program in memory
- FOR X = Y TO Z makes the VIC repeat the same part of the program several times. X has the value Y to begin with, and is increased by 1 each time until it becomes equal to Z
- using RND(1) gives a random number between 0 and 1 (almost)
- using INT chops off the figures to the right of the decimal point, leaving an integer (whole number). *Note:* this is true for positive numbers only; more later
- GOTO makes the VIC go next to the program line numbered in the GOTO command.

Above all you have learnt that computing is fun!

Chapter Four
Back to Basics

Before going on to discover the excitement of the VIC sound effects, we will look more closely at the way the computer handles numbers. After all, handling numbers is something that computers are supposed to be good at. Switch on, and set the micro the task which is usually regarded as the ultimate in arithmetical prowess. Type:

```
PRINT 2 + 2
```

As might be expected, the answer, 4, is printed immediately you press RETURN. Incidentally, do not forget the warning about the *shifted* '+' in Chapter Three. Since we shall be doing a lot of PRINTing in this chapter, it is as well to learn the short form of this command. In common with many other computers the VIC 20 accepts a '?' as the short form of PRINT. We will still use the long form in the text, but on the screen you can type:

```
? 2 + 2
```

and get the answer 4. Nothing happens until you press RETURN of course. But this has probably become a habit with you. From this point on, there will be no more reminders about pressing RETURN when you want the micro to take action.

Try some other additions, such as:

```
PRINT 2 + 4 + 6
PRINT 2 + 4 + 6 + 8 + 1ø
PRINT 2.5 + 4.7
```

Yes, numbers with decimal places can be handled just as well as integers (whole numbers). To type a decimal point we use the full stop (period). You can have up to nine digits (figures) in the number:

```
PRINT 1.23456789 + 9.87654321
```

Try this, and note that the answer has nine digits too.

Operators

The '+' sign is called an *operator*. Placed between two numbers, it causes an operation to be performed on them. The operation caused by '+' is *addition*. Try the effects of these three other operators:

```
PRINT 6 - 4
PRINT 6 * 4
PRINT 6 / 4
```

The answers confirm that '–' is the operator for *subtraction*, '*' is the operator for *multiplication*, and '/' is the operator for *division*.

Now type:

```
PRINT 4/6
```

This is a division which has an infinite number of decimal places in its result. The real answer is '0.666 666 666 666 666 666...' and so on, for ever. The micro can print no more than nine digits in its answer, so the result it gives is:

```
.666666667
```

Note that if the figure which goes before the decimal point is zero, no figure is printed. The same rule applies when you are typing in a number. If you want to type 0.456, for example, there is no need to type the zero; just type '.456'. Note also that the last figure has been 'rounded off' to the nearest value. If you get the computer to divide 100 by 6, the answer given is '16.6666667'. There are still nine digits. The result is rounded off, as before.

Priorities

When an expression such as '2 + 4 – 5' is being worked out, the operators are read from left to right, just as if we were saying: 'Two plus four makes six, take away five leaving the answer, one'. Now try this:

```
PRINT 2 * 4 + 6
```

Here we might say: 'Two times four is eight, and add six to get the answer, fourteen'. The computer also gets the answer 14. But now try this:

```
PRINT 2 + 4 * 6
```

However, 9 decimal places is near enough to the value of 1/3 to give the correct answer. But if we are lazy and enter:

```
PRINT 8↑ .333
```

we get an answer showing that .333 is *not* close enough to the true value of 1/3. Try this and see what you get. There is one way around the problem of such awkward fractions. Here it is:

```
PRINT 8↑ (1/3)
```

This has the same effect as entering all 9 threes, but requires far less typing. It leads to another rule, that the VIC begins any calculation by first working out the values of expressions or parts of expressions which are in brackets. The value of 1/3 is first worked out by the computer, giving 0.333333333, and then this value is used to calculate 8↑ .333333333.

The brackets rule applies to other calculations too. Compare the different results you get with:

```
PRINT 8 + 2 * 4 + 3
PRINT (8 + 2) * (4 + 3)
```

In the first expression, multiplication takes priority. In the second expression the additions have priority, because they are inside the brackets. The multiplication comes next. This shows how we can alter the order in which operators are read by using brackets. There is a summary of all the rules at the end of this chapter.

Numbers great and small

This section is for the mathematically minded. If you would rather miss it out, go on to the section on Variables. What is the biggest number that the computer can handle? Let's find out. We test if it can handle a given number by asking it to PRINT it:

```
PRINT 1ØØØØØØ
PRINT 1ØØØØØØØ
PRINT 1ØØØØØØØØ
```

No problems so far – not surprising, for we have said that the VIC can handle numbers with nine digits. What about ten digits?:

```
PRINT 1ØØØØØØØØØ
```

You might think that it is cheating to turn over to a new method of

This gives the correct result '2', which is twelve divided by two (giving 6) and then divided again, by 3. Although you used '×' in the original expression, you use '/' instead, in the line typed into the computer.

Another operator

There is a fifth arithmetical operator, with the symbol '↑'. Type this line and see if you can discover what '↑' does:

```
PRINT 3↑2
```

The result is '9', which is 3 times 3. In other words, *two* threes multiplied together. The mathematical way of writing this is 3^2, which is read as '3 to the power 2'. You can also call it 'three squared'. Here are some more examples to try:

```
PRINT 3↑4
PRINT 5↑2
PRINT 2*3↑2
```

In the last example you find that the answer is 18, or twice 9. The computer is reading '↑' first, so obtaining 9, then multiplying the result by 2. The '↑' (or *exponentiation*) operator takes priority over '*' and '/'.

This operator does not necessarily need an integer as the 'power':

```
PRINT 2↑2.5
```

gives 5.65685425 as the value of $2^{2.5}$. You will not be able to check this by working it out in your head, but it is easy to see that it must come between 2^2 (= 4) and 2^3 (= 8). You may recall that to take a number to the power 0.5, is the same thing as taking its square root. Here are two examples:

```
PRINT 9↑.5
```

This gives the square root of 9, which is 3. To find the cube root of 8, use this:

```
PRINT 8↑ .333333333
```

This gives the expected answer 2. There is a difficulty here, that the fraction 'one-third' really has an infinite number of decimal places; after the nine 3s we have typed, there should be many more.

However, 9 decimal places is near enough to the value of 1/3 to give the correct answer. But if we are lazy and enter:

```
PRINT 8↑ .333
```

we get an answer showing that .333 is *not* close enough to the true value of 1/3. Try this and see what you get. There is one way around the problem of such awkward fractions. Here it is:

```
PRINT 8↑ (1/3)
```

This has the same effect as entering all 9 threes, but requires far less typing. It leads to another rule, that the VIC begins any calculation by first working out the values of expressions or parts of expressions which are in brackets. The value of 1/3 is first worked out by the computer, giving 0.333333333, and then this value is used to calculate 8↑ .333333333.

The brackets rule applies to other calculations too. Compare the different results you get with:

```
PRINT 8 + 2 * 4 + 3
PRINT (8 + 2) * (4 + 3)
```

In the first expression, multiplication takes priority. In the second expression the additions have priority, because they are inside the brackets. The multiplication comes next. This shows how we can alter the order in which operators are read by using brackets. There is a summary of all the rules at the end of this chapter.

Numbers great and small

This section is for the mathematically minded. If you would rather miss it out, go on to the section on Variables. What is the biggest number that the computer can handle? Let's find out. We test if it can handle a given number by asking it to PRINT it:

```
PRINT 1ØØØØØØ
PRINT 1ØØØØØØØ
PRINT 1ØØØØØØØØ
```

No problems so far – not surprising, for we have said that the VIC can handle numbers with nine digits. What about ten digits?:

```
PRINT 1ØØØØØØØØØ
```

You might think that it is cheating to turn over to a new method of

expressing numbers, but this is what happens. The expression '1E + ∅9' has appeared on the screen. This is the computer's way of saying:

$$1 \times 10^9$$

An expression written out like this is said to be in *standard form* or *exponential form*. This is the way that large numbers can be typed in without the need for long rows of figures. It certainly helps in the quest to find out which is the largest number the computer can handle. See what the computer makes of these:

```
PRINT 1E + 1∅
PRINT 1E + 2∅
PRINT 1E + 3∅
PRINT 1E + 4∅
```

It is only the last of these which proves too large for the computer. At the last one, it replies with the message: "?OVERFLOW ERROR". You can narrow the search by trying:

```
PRINT 1E + 31
PRINT 1E + 32
```

and so on. The largest number is between 1E + 38 and 1E + 39. 1E + 38, or 1×10^{38}, is '1' followed by 38 zeros, or a hundred million million million million million million. It is not often that we shall be wanting the VIC to handle numbers as large as this. If you really want to know what the largest number is, you can narrow the search still further by trying:

```
PRINT 1.1E + 38
PRINT 1.2E + 38
```

and so on. Having established the first decimal place, you can narrow the search to the second decimal place. Continue to the third decimal place and as far beyond that as you like.

For all practical purposes the largest positive number is 1E + 38. The largest negative number is −1E + 38. The smallest possible number (that is to say, the one closest to zero that is not so small that it *is* zero) can be found in a similar way. Here we use a minus sign in the standard form. If you type:

```
PRINT 1E − 1
```

the display does *not* repeat the number in the standard form. Instead you see the ordinary decimal form, ∅.1. The expression 1E − 1, means 1×10^{-1}, which can also be written as:

$$\frac{1}{10^1}, \text{ which is 0.1 in decimals.}$$

Similarly, 1E − 2 means 1×10^2, which can be written as:

$$\frac{1}{10^2} = \frac{1}{100}, \text{ which is 0.01 in decimals.}$$

The computer replies to 'PRINT 1E − 2' with '0.01' but beyond that it uses standard form. When you type:

```
PRINT 1E − 3
```

it replies with '1E − 03'. What is the smallest number that it can handle?

Variables

So far we have been typing numbers which are dealt with and promptly forgotten. If we want a number to be remembered for use on some future occasion, we must say to the VIC: 'Here is a number I want you to remember. To help remind you which number I'm talking about, I'll give it a name. The number is 4 and its name is JIM. When I ask you about JIM, remember that JIM means 4'.

A computer does not really understand this way of talking (though maybe some computers will be able to do so in the future!), but we get the same effect by typing:

```
JIM = 4
```

Has the computer done as asked? Let's find out by typing:

```
PRINT JIM
```

The answer comes back immediately – the number 4 is printed. Ask the computer to print JIM again, and you will find that it does not forget. The only time it forgets is when the power is switched off.

Having been given this number named JIM, the micro can use the name just like number. Try some of these:

```
PRINT JIM + 5
PRINT JIM + JIM
PRINT 2 * JIM
PRINT JIM↑2
```

Printing JIM-squared sounds odd, but it can be done!

We can ask the computer to remember two, three and many more numbers at the same time, each with its own name. Let's give it another one to work with:

```
KEN = 5
```

Then try:

```
PRINT JIM + KEN
PRINT 2 * KEN - JIM
```

and any other expressions you like.

Since RAM can be changed, we can ask the computer to change any number to a different one. We simply issue a new instruction:

```
JIM = 99
```

and, from then on, the fact that JIM once was 4 is forgotten and JIM becomes 99. Now we get a new set of answers when we type:

```
PRINT JIM + KEN
PRINT 2 * KEN - JIM
```

Find out what happens when you type:

```
JIM = KEN
```

Words such as JIM and KEN, which we use to name the numbers we want the micro to remember, are called *variables*. This is a good name for, as we have seen, they can be varied. The VIC can remember many variables, but these do not all have to be the names of one's friends. For example, you could type in names and prices of accessories for the VIC:

```
JOYSTICK = 13
LIGHTPEN = 25
RAMPACK = 2Ø
TAPE = Ø.5Ø
```

Then if you want to find out the total cost of the things you buy, you type:

```
PRINT JOYSTICK + RAMPACK + TAPE
```

or whatever other combination of goods you are buying, and you get the total cost.

Using names as variables helps to make your programs easier to read and understand, but unfortunately it takes up more memory space. There is one other point to watch out for. The computer does

not remember the whole of the name, but only the first two letters. So if you tell it:

```
JIM = 5
```

and then ask it to

```
PRINT JI
PRINT JIG
PRINT JITTERS
```

it replies with 5 every time. Thus a variable is defined by its *first two letters*, but it need have only one letter if you wish. The first letter of any variable must be a letter, but after that you can use a letter or a numeral. These are the valid variables which begin with J:

```
J
JA, JB, JC, ... JZ
J1, J2, J3, ... J0
```

There is a similar set for all 26 letters of the alphabet. If you want to use JIM and KEN, these are acceptable, but the computer thinks of them only as JI and KE.

A few combinations of letters are not allowed as variables. TO, ON, OR, GO, and IF are BASIC words. If you try to use these as variables, you will get an error message. There are also the special variables ST and TI which are used by the computer (see later in this chapter). Longer names such as TOMATO, or ORANGE, which have a BASIC word as their first two letters must not be used as variable names. Similarly, you cannot use words which have BASIC words at the beginning. Examples are FOREST, POKER, and RUNCIBLE.

Variables can be used in many ways in programs. We have already seen this in the Magic Carpet program (Listing 3.3). There we used single-letter variables:

C for the colour of the disc (line 2Ø)
X for the positions of the rows of discs (line 3Ø)
Y for the positions of the columns of discs (line 1ØØ)
J for counting how many discs had been printed in a row (lines 5Ø to 8Ø) or column (line 11Ø to 14Ø). J was then used for telling the computer where to POKE the Character Code RAM and Colour Code RAM.

Although the variables all had integer (whole number) values in that program (e.g. Ø, 22, 7777, 38497), we have seen that a variable

can also be given values which are non-integers (e.g. 2.45, 12.54). Negative values and numbers in standard or exponential form (e.g. 1.23E + 06) can also be given to variables.

Integer variables

The variables described above are *numeric variables* (sometimes called *floating-point* variables). There is another kind of variable called an *integer variable.* This cannot be given non-integer values. We define a variable as an integer variable by typing a '%' after its name:

```
JIM% = 6
```

Even if you try to make JIM% a non-integer, the computer ignores anything after the decimal point. Try this:

```
JIM% = 7.89
PRINT JIM%
```

If you still have JIM (the numeric variable) in memory, ask the printer to PRINT JIM and then PRINT JIM%. You will see that JIM and JIM% are distinct variables and can hold different values. Using integer variables gives us a whole new set of variable names, should we need that many different variables all at once.

We could have used integer variables in the Magic Carpet program, but there was little point.

The main advantage of integer variables is that they take less memory to store. If you have lots of numbers and little memory, using integers can be helpful. It also takes the computer less time to look up the value of an integer variable so, if you want a program to run as fast as possible, use integers (assuming you do not want any decimal places in your numbers). But, for general use, the need to type in the '%' after each name is a nuisance, and we prefer to use ordinary numeric variables.

There is another disadvantage of integer variables, that the number cannot be greater than +32767 or less than −32768. This range is large enough for very many purposes, but not for all. You can use integers in all the usual ways, and mix them in expressions with the ordinary numeric variables.

Keying in values

In this chapter you have typed variables and their values into the computer and pressed RETURN so that the computer has worked on them immediately. As we have seen with the Magic Carpet program, variables are frequently used in programs too. In Listing 4.1 there is a program which lets you change the colour of a PRINTed pattern.

```
10 PRINT"□"
20 C = 2
30 PRINT"□"
40 POKE 36879,220
50 FOR J = 7680 TO 8205 STEP 5
60 POKE J,88
70 POKE J + 30720,C
80 NEXT J
90 FOR K = 1 TO 2000:NEXT K
100 POKE 36879,27
110 GOTO 10
```

Listing 4.1. Changing the colours in a pattern.

Run the program and see the pattern appear in red. Now change the value in line 2Ø to make the program give a yellow pattern instead. Table 7 (Appendix A) lists the codes you need.

The snag with this program is that you have to alter the program line every time you want a different colour. It would be better to be able to change the colour by simply keying in a number at the keyboard. Type in this new line 2Ø:

```
2Ø INPUT C
```

Now RUN the program. No pattern appears, but instead you see a question-mark. The VIC is waiting for you to enter a number. For a the moment, the program is halted. Key a number between Ø and 7, depending on what colour you want the pattern to be. Press RETURN. The variable C is given the number you have just keyed in. The program continues, using this value for C. The pattern appears on the screen in the desired colour. INPUT is a very useful command. We can improve it by making it print a message:

```
2Ø INPUT "WHAT COLOUR CODE";C
```

Note the semicolon. Now the user of the program can see what the computer is expecting to be keyed in. More about INPUT later.

String variables

Here is yet another set of variables at your disposal. These all have the '$' sign at the end of the variable name. A string variable is just a string of characters – letters, numbers, graphics symbols, punctuation marks and the like. For example:

```
NAME$ = "ALGERNON BLOGGS"
DATE$ = "29 FEBRUARY 1985"
```

Inform the computer of these string variables by typing them in as above. Now type:

```
PRINT NAME$; DATE$
```

The computer PRINTs out Algernon's full name and birthday. Note that we have told the computer to PRINT two strings, one after the other. The items in the list of strings to be PRINTed are separated by a semicolon (;). Unfortunately, the display is spoiled because the printer does not put a space between the strings when it PRINTs. To get the space we could either type a space at the end of one of the strings (NAME$ = "ALGERNON BLOGGS ") or add the space to our list of strings to be printed:

```
PRINT NAME$;" ";DATE$
```

When we tell the computer what is to be in a string variable, we use double quotation marks (") at both ends. Once a string has been defined, we can just use its name.

Obviously there are things we can do with numeric and integer variables which we cannot do with string variables. We cannot, for example, use expressions such as JIM$ = 3 * JOE$ + JAN$↑JON$. The only operator that works with strings is '+'. If you still have NAME$ and DATE$ in memory, type in this line:

```
J$ = NAME$ + "  " + DATE$
```

Then type:

```
PRINT J$
```

Your first command tells the computer to add NAME$, the space and DATE$ together and call the result J$. Adding strings together simply means joining them together, in the order they are listed.

One important point is that a string such as "123" which holds the number 123 is *not* the same as the numeric or integer variable 123. You can say:

```
PRINT 2 * 123
```

and get the answer 246, but you cannot do this with a string. To illustrate the difference between strings variables and the other types, try this:

```
A$ = "123"
B$ = "456"
PRINT A$ + B$
```

This does *not* give the sum 579, but "123456". There are lots of other things we can do with string variables. These will be explained in Chapter Eight.

Special variables

There are three variable names which are reserved for use by the computer. These are ST, TI and TI$. We shall not be concerned with ST, but TI and TI$ are extremely useful. Among the many circuits that make up the computer there is one which generates a pulse sixty times a second. This is known as the *clock*. The pulses are counted and the result stored as the variable TI. As you can guess, this is short for TIME. Use either TI or the whole word TIME to read this variable:

```
PRINT TI
```

The result you get depends on how long ago you switched on the computer. The value is the number of sixtieths of a second that have passed since you switched on. Putting it another way, TI divided by 60 gives the number of seconds.

TI$ is a string consisting of six figures. The two on the right give the number of seconds since the computer was switched on, the two in the middle give the number of minutes and the two on the left give the number of hours. So if you type:

```
PRINT TI$
```

and the result is "ø13523", you know that the computer has been on for 1 hour, 35 minutes and 23 seconds.

Although TI and TI$ automatically start from zero when the computer is switched on, it is possible to reset them to zero whenever you want. TI cannot be reset directly but, when TI$ is reset, TI is reset too:

```
TI$ = "øøøøøø"
```

All six figures must be made zero. In this way TI and TI$ can be used in games and other programs, as we shall see later. If you want to, you can set TI$ to any other time. For example, 'TI$ = "123456"' sets it to 12 hours, 34 minutes and 56 seconds. This is handy if you have a program which needs the time of day. TI is reset to zero only when TI$ is reset to zero. Setting TI$ to other times does not change TI.

Listing 4.2 shows a short program which shows how TI can be used. It is a reaction timer.

```
10 PRINT"□"
20 POKE 36879,122
30 FOR J = 1 TO INT(RND(1) * 10000):NEXT
 J
40 POKE 7910,127
50 POKE 38630,2
60 TI$ = "000000"
70 GET A$
80 IF A$ = "" THEN 70
90 PRINT"□"
100 POKE 36879,47
110 PRINT"■YOUR REACTION TIME WAS":PRINT
:PRINT TI/60;"SECONDS"
120 GOTO 120
```

Listing 4.2. Reaction timer.

After typing PRINT" in line 11Ø press CTRL with 2. This gives a reversed 'E' on the screen which the computer interprets as "print the letters in white". RUN the program. The screen is cleared in line 1Ø. Line 2Ø makes the screen red with a yellow border. There is then a wait for a random length of time, after which a character appears at the centre of the screen (line 4Ø, 5Ø). The program resets TI (by resetting TI$) as soon as it has printed the character. When you see this character appear, press any key. As soon as you press the key, the program clears the screen and reverses the screen and border colours. It then prints out your reaction time, using TI. The value of TI divided by 6Ø is your reaction time, in seconds.

An interesting feature of this program is the use of the command GET. Like INPUT, this is a way of allowing the user at the keyboard to interact with a program while it is running. There are three main differences between INPUT and GET:

(1) *GET does not wait.* When the computer reaches the line in which GET occurs, the keyboard is read. The variable which is being got (i.e. A if the command is GET A or A$ if the command is GET A$) takes the value of any key which is being pressed *at that moment*. If no key is being pressed, then the variable takes a null value. If the variable is numeric (e.g. A) its value becomes zero. If the variable is a string variable (e.g. A$), it becomes an empty string ("").

(2) *GET works without you pressing RETURN.* This follows from what has just been said. By contrast, INPUT waits for you to type in as much as you want; you press RETURN when you have finished.
(3) *GET can accept only one character.* This follows from the above, for only one key-press can be recognised by the computer at one time.

In the reaction program we use GET A$. This allows the user to press *any* key. If we had used GET A, then a number key must be pressed – if another key is pressed the result is an error message! When the program reaches GET A$, and if no key is being pressed, A$ becomes an empty string. Line 8Ø checks if A$ is empty or not. If it *is* empty, (IF A$ = "") then the program goes back to read the keyboard again. If it is not empty (a key has been pressed) the program goes on to the next line to begin timing your reaction.

Another interesting feature of the reaction program is the *branch* at line 8Ø. If a key is pressed, the program continues with the timing. If no key is pressed, it goes back to check the keyboard again. This branching is the result of using the sequence of words:

IF (condition) THEN (action)

The condition is 'A$ = ""'. If this condition is *true* (that is, if A$ is an empty string), then the computer takes the action specified. Here the action is 'GOTO 7Ø'. So if A$ is an empty string, the computer goes back to line 7Ø to test the keyboard again. It continues doing this for as long as the condition remains true. If the condition is not true (that is, if A$ is not an empty string), the computer does not take the action specified, but simply goes on to the next line of the program. 'IF ... THEN ...' is a sequence of words we shall be using a lot in other programs in this book.

To summarise

In this chapter, you have found out how to:

- get the VIC to do all kinds of calculations
- work with very large and very small numbers
- use variable names
- use numeric variables, integer variables and string variables
- measure time, using the variables TI and TI$
- use the INPUT and GET commands
- write a branching program, using IF (condition) THEN (action)

You have learned that:

- the arithmetic operators are +, −, *, /, and ↑
- the rule of priority in calculations is:

  First, expressions inside brackets
  Then, ↑
  Then, * and / (left to right)
  Last, + and −.

Chapter Five
Sounds Fantastic!

Chapter Three showed that the Video Interface Chip (from which the VIC gets its name) enables us to achieve the most elaborate visual effects with the minimum of programming. We shall find out a lot more about this aspect of the chip in Chapter Seven, but here we explore the other side of its nature – the sound generators.

The VIC has three tone generators and one noise generator, which may be independently controlled. They generate signals which are added together and fed to the audio circuits of the TV set. If you have turned down the volume of the TV to avoid the hissing sound that it emits when first switched on, turn the volume up again now, ready for the first demonstration program which is shown in Listing 5.1.

```
10 S = 36875
20 V = 36878
30 INPUT"FREQUENCY,VOLUME";P,Q
40 POKE S,P
50 POKE V,Q
60 FOR K = 1 TO 500
70 NEXT K
80 POKE S,0
90 POKE V,0
100 GOTO 30
```

Listing 5.1. Demonstrating how the sound generators work.

Before RUNing this program, look through it to see what it does and how it operates. Lines 1ϕ and 2ϕ set the values of two variables, S and V. These are the addresses in RAM of two sections of the sound chip. S is the address of one of the three tone generators (Table 8 in Appendix A). V is the address of the volume control register. This varies the volume of sound coming from all four generators. It is not possible to vary them individually. As Table 8 shows, the frequency of the sound coming from the generators depends on the value POKEd into their addresses. This value must be between 128 and 254. The higher the value, the higher the

frequency pitch. POKEing any value between Ø and 127 turns the generator off, while POKEing 255 gives the same note as 128.

The volume control requires a number between Ø and 15. The lower the number the lower the volume, Ø giving silence. POKEing Ø to the volume control is not quite the same as POKEing Ø to a sound generator. Setting the *volume* to Ø does *not* turn off the generators. If the volume is increased again, the generators are heard sounding, just as they were before. By the way, if the sound ever becomes too horrible to bear, there is an easy way out! Switch all sound generators off and reduce volume to Ø by pressing RUN/STOP with RESTORE.

We are using variables S and V in this program so that you may be clear about which addresses are being POKEd at lines 4Ø and 5Ø. The program allows you to type in two numbers, one for frequency (128 to 254) and one for volume (Ø to 15). Line 3Ø uses the INPUT statement to take in two numbers. Note how the line is made up:

The word *INPUT*,

followed by a *message*, if any, between quotation marks (sometimes called a 'prompt' as it prompts the user about what to key in),

followed by a *semicolon* (;), which is omitted if there is no message,

followed by a *list* of one or more variables, separated by commas; the variables in the list can be mixed, including numeric, integer or string variables in any order.

When this line of the program is reached, the computer prints the prompting message, asking you to key in frequency and volume. Key in *two* numbers (in their correct ranges) separated by a comma. Then press RETURN.

If you forget, and press RETURN after the first number, the computer prints '??' indicating that it is still waiting for the second number. Key it in and press RETURN again. Since this INPUT statement has two *numeric* variables in it, you must be sure to type in *numbers*, not letters. If you type letters by mistake the computer gives you a second chance by printing 'REDO FROM START' and repeats the message and question mark.

Lines 4Ø and 5Ø POKE the values you have just set, and the sound begins. Lines 6Ø and 7Ø make the computer count from 1 to 5ØØ. This is a *delay loop*. It is a way of delaying the computer for a while so that it does not rush on to turn the sound off at line 8Ø. It gives you about one second in which to hear the sound.

It is important to switch off the sound at the end of a program. This is done in lines 8ø and 9ø. After this, the micro is sent back to line 3ø to ask you to key in details of the next sound you would like to hear.

RUN this program and try it for several different frequencies and volumes. Table 9 in Appendix A tells you which values give particular notes on the musical scale. This program is using the Tenor tone generator, so refer to the middle column of figures. The values in the table are the nearest integer, for the generators can accept only integer values. Certain of the notes may be slightly out of frequency for this reason, though the difference is usually not apparent except to the experienced musician.

Table 9 also shows the values used with the other tone generators. To try these out, change line 1ø to:

1ø S = 36876

for the Treble tone generator, or to:

1ø S = 36874

for the Bass tone generator.

The fourth sound generator is different from the other three. Instead of a tone, it produces a noise. This is technically known as 'white noise'. The output from this generator is filtered so that its sound can vary from that of a roaring mountain torrent (P = 128) to that of escaping steam at high pressure (P = 254). Alter line 1ø of the program to:

1ø S = 36877

and put the noise generator through its paces.

Changing frequency and volume

Sounds usually do not consist of a simple burst of tone or noise at constant frequency and volume. The frequency may rise or fall and so may the volume. Try the program in Lising 5.2.

```
10 S = 36875
20 V = 36878
30 POKE V,15
40 FOR J = 128 TO 254
50 POKE S,J
60 FOR K = 1 TO 10: NEXT K
70 NEXT J
80 POKE S,0:POKE V,0
```

Listing 5.2. Making the frequency rise or fall.

Here we POKE the tone generator repeatedly by means of a loop. As J increases from 128 to 254 the frequency rises. There is a very short delay each time the computer goes round the main loop. This is caused by the delay loop of line 6ϕ. This is a good example of *nested loops*. The outer loop (FOR J = 128 TO 254 ... NEXT J) encloses the inner loop (FOR K = 1 TO 1ϕ ... NEXT K). When loops are nested, the *first* variable to be used as a loop counter (J, in this example, at line 4ϕ) is also the *last* to be used (NEXT J, at line 7ϕ).

You can alter line 4ϕ to make the note fall in frequency:

```
4ϕ FOR J = 254 TO 128 STEP −1
```

The loop is counting down this time. To tell the computer which way to count, the additional word 'STEP' is used. Counting backward by '1's is indicated by 'STEP −1'. This word can be used with other values. If you want the tone to rise quickly, line 4ϕ could become, for example:

```
4ϕ FOR J = 128 TO 254 STEP 1ϕ
```

J then takes the values 128, 138, 148, ... up to 248. The next step after 248 would be 258 but, since this is higher than the upper limit specified, the loop does not repeat with this value. Try this in the program to see what it sounds like. Also try varying the beginning and ending values given in line 4ϕ. Try the effects of different negative STEP values. In a 'FOR ... NEXT' loop with no 'STEP', the value for step is automatically taken as +1.

Now for some more effects that can be obtained by simple modifications of the program. Type in this extra line:

```
75 GOTO 4ϕ
```

This sends the micro back to repeat the loop. If the STEP in line 4ϕ has a value about 25, you get a sound rather like that of a heavy motor running. Remember about RUN/STOP pressed with RESTORE when you tire of this sound!

The program in Listing 5.3 has some features in common with the previous program (Listing 5.2).

Line 1ϕ shows that this program uses the treble tone generator and the noise generator (N = 36877). The first section produces a note which falls in frequency (line 3ϕ takes care of this). At line 5ϕ we see an additional operation within the loop. Not only does the note fall in frequency, but it gets louder. It starts at volume ϕ, when J is 23ϕ, because (23ϕ−J)/4 = ϕ/4 = ϕ. Note the use of brackets to make the computer do the subtraction (23ϕ−J) *before* it does the division.

```
10 S = 36876: N = 36877
20 V = 36878
30 FOR J = 230 TO 180 STEP - 1
40 POKE S,J
50 POKE V, (230-J)/4
60 FOR K = 1 TO 30:NEXT K
70 NEXT J
80 POKE S,0
90 POKE V,15
100 POKE N,160
110 FOR K = 1 TO 250: NEXT K
120 FOR J = 15 TO 1 STEP -1
130 POKE V,J
140 FOR K = 1 TO 200:NEXT K
150 NEXT J
160 POKE V,0
170 POKE N,0
180 END
```

Listing 5.3. The falling bomb is the first stage of the Air Raid program.

By the end of the loop, when J = 18ø, the value being POKEd to the volume control is (23ø − 18ø)/4 = 5ø/4 = 12.

The overall effect of a tone of descending pitch and increasing volume is highly suggestive of a descending bomb, which is why this program is called Air Raid. In line 8ø the tone generator is switched off. Line 9ø turns the volume fully up, and line 1øø starts the noise generator. This gives the beginning of the explosion. This volume is held for about half a second (line 11ø). Then the volume is reduced in stages to zero, as the echoes of the explosion die away (12ø − 15ø). Finally (lines 16ø and 17ø) all is silence as the volume is reduced to zero and the noise generator is switched off.

This program illustrates the idea of 'envelope'. The envelope of the sound of the falling bomb is simple – it increases steadily. This is seen in Fig. 5.1, which shows the sequence of the effects. Drawing diagrams such as this is very helpful when you are planning

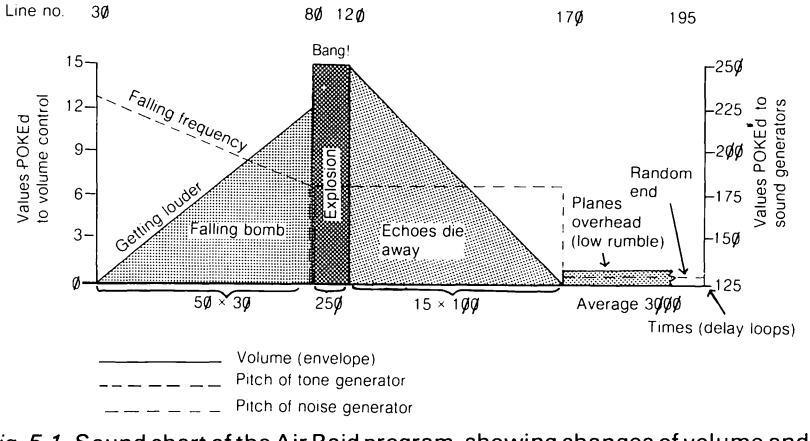


Fig. 5.1. Sound chart of the Air Raid program, showing changes of volume and frequency.

programs which have sound. The envelope of the explosion is more complicated, since the sound level is steady at first, then dies away.

If you want a continuous air raid, add these lines to the program:

```
18ø FOR K = 1 TO INT(RND(1) * 6øøø)
19ø NEXT K
2øø GOTO 3ø
```

This gives a delay of variable length, decided by which number is selected randomly. After this delay the next bomb falls.

To make the illusion completely audio-visual, here are some more lines to add to the program:

```
5 PRINT "♥": POKE 36879,8
```

This clears the screen and makes the screen and border black (this is a night raid!).

```
135 POKE 36879,9
14ø FOR K = 1 TO 1øø: NEXT K
145 POKE 36879,24
147 FOR K = 1 TO 1øø: NEXT K
```

This splits the original delay at line 14ø into two halves. Lines 135 and 145 alternately make the screen white with black border, then black with white border.

```
165 POKE 36879,8
```

makes the screen and border black again. For the best effect, turn the room lights low. Run the program. Now the most spectacular flashes emanate from the screen each time a bomb falls. Still one thing missing? Yes, of course, the sound of the bombers. Change these lines:

```
16ø POKE V,1
17ø POKE N, 128
```

and add this line

```
195 POKE V,ø: POKE N,ø
```

Now you can hear the drone of the bombers overhead between the sounds of falling bombs and explosions. Listing 5.4 shows the Air Raid program with all these additions. What about adding a few stars to the night sky? This and other improvements are easy to make by typing in one or two more lines. There really is no end to the versatility of the VIC.

```
5 PRINT"□":POKE 36879,8
10 S = 36876: N = 36877
20 V = 36878
30 FOR J = 230 TO 180 STEP - 1
40 POKE S,J
50 POKE V, (230-J)/4
60 FOR K = 1 TO 30:NEXT K
70 NEXT J
80 POKE S,0
90 POKE V,15
100 POKE N,180
110 FOR K = 1 TO 250: NEXT K
120 FOR J = 15 TO 1 STEP -1
130 POKE V,J
135 POKE 36879,9
140 FOR K = 1 TO 100: NEXT K
145 POKE 36879,24
147 FOR K = 1 TO 100: NEXT K
150 NEXT J
160 POKE V,1
165 POKE 36879,8
170 POKE N,128
180 FOR K = 1 TO INT(RND(1)*6000)
190 NEXT K
195 POKE V,0:POKE N,0
200 GOTO 30
```

Listing 5.4. The completed Air Raid program.

All systems go

Already we have had a tone generator and the noise generator operating together. The sound of music can be much improved by calling all three tone generators into action at once. But before we can go on to do this, in the Carousel program (Listing 5.7), there is something to be learned about the way the computer handles data.

If many notes are to be played in quick succession, and up to three are to be played at once, it is not possible for you to sit at the keyboard feeding in the numbers, as you did with the program in Listing 5.1. The numbers (or data) must be inside the computer already, waiting to be used in the program. One way of storing data in the computer is in a DATA statement. Listing 5.5 is a program to show how this works.

Lines 1ØØ and 11Ø are separate from the main program, as can be seen from the fact that line 9Ø says 'END'. In these lines we have the values that are to be POKEd into the address of the Tenor tone generator. The DATA statements contain 31 numbers, one for each note of the tune. Lines 3Ø to 7Ø are a loop which is repeated 31 times, once for each note. The command READ N causes variable N to take one of the values in the data statement. It takes a different value each time round the loop.

The first time round, for example, N takes the value 193. At line 5Ø this is POKEd to the Tenor tone generator. Middle C is heard

```
10 S = 36875: V = 36878
20 POKE V,15
30 FOR J = 1 TO 31
40 READ N
50 POKE S,N
60 FOR K = 1 TO 200:NEXT K
70 NEXT J
80 POKE S,0:POKE V,0
90 END
100 DATA 193,206,200,209,206,214,206,193
,0,193,206,200,209,206,193,0
110 DATA 193,206,200,209,206,214,206,193
,0,218,200,209,206,193,0
```

Listing 5.5. Storing the notes of a tune in a DATA statement.

from the loudspeaker. The next time round the loop, N takes the *second* value of line 1ØØ. This value, 2Ø6, is POKEd to the generator, giving the note of E. Thus each value is read in turn. Where there is a 'Ø' we have a period of silence. The fact that the data are on two separate lines is not important. READing continues from one line to the next, just as if all data were on one line. The computer READs each item of data in turn, playing each note as it does so. After 31 notes (including silence as a 'note'), the tune is complete and the program is over.

Now try adding this line:

```
85 GOTO 2Ø
```

When you RUN the program you hear the tune once, but at the end of it you see '? OUT OF DATA ERROR IN 4Ø' on the screen. All 31 items of data were read the first time the tune was played. Now there are no more items to be READ. Fortunately, there is a way to send the computer back to the start of the list:

```
83 RESTORE
```

This command makes the computer read from the beginning of the list again, so you now hear the tune repeated as many times as you want. RESTORE can be placed at any point in a program. Even if the computer has not READ to the end of a set of DATA statements, RESTORE sends it back to start from the beginning again.

Arrays

Another way of holding data in a program is to put it into an *array*. This, like a DATA statement, is a kind of list. An array holds a set of variables all of which have the same name. Just as you can have a variable named JIM, so you can have an array called JIM. A

variable JIM can have only one value at a time, but an array JIM can have may different values at once.

We set up an array by 'dimensioning' it. In other words we decide what size it is to be – how many variables we want it to hold. For example if we want JIM to be an array which will hold 10 different values all at once, we type in the line:

```
1Ø DIM JIM(1Ø)
```

If you still have the previous program on the computer, get rid of it by typing NEW, then type in the line above. Nothing much seems to happen when you press return but the VIC has set aside a part of its memory ready to hold the array of ten variables. The fact that the name JIM is followed by a number or numbers in brackets tells the computer than JIM is an array, not just a numeric variable. If you want, you can have a numeric variable JIM in existence at the same time as the array JIM, and the computer treats them separately. The rules about the names of variables apply to the names of arrays too. JIM(1Ø) holds numeric variables, JIM%(1Ø) holds integer variables, while JIM$(1Ø) holds string variables. You cannot mix variables of different types in the same array. But all these three arrays named JIM can exist at the same time, though probably it would be more confusing to *you* than to the computer to have them all with such similar names! The other rules for names are as already set out in Chapter Four.

We can think of the array we have just dimensioned as a list, set out as in Fig. 5.2. Each item in the list has a number, its *subscript*. You will notice that the array can actually hold 11 numbers, not 10, for the list begins with subscript zero. Usually we forget about subscript zero, and proceed as if the list really started with subscript 1. Now to put some numbers into the array:

```
2Ø FOR J = 1 TO 1Ø
3Ø INPUT JIM(J)
4Ø NEXT J
5Ø FOR J = 1 TO 1Ø
6Ø PRINT JIM(J)
7Ø NEXT J
8Ø END
```

This program is in two halves. The first half takes data from the keyboard as you key it in, and puts it in array JIM (to save the long-winded use of the phrase 'array JIM' we will use 'JIM()' from now on).

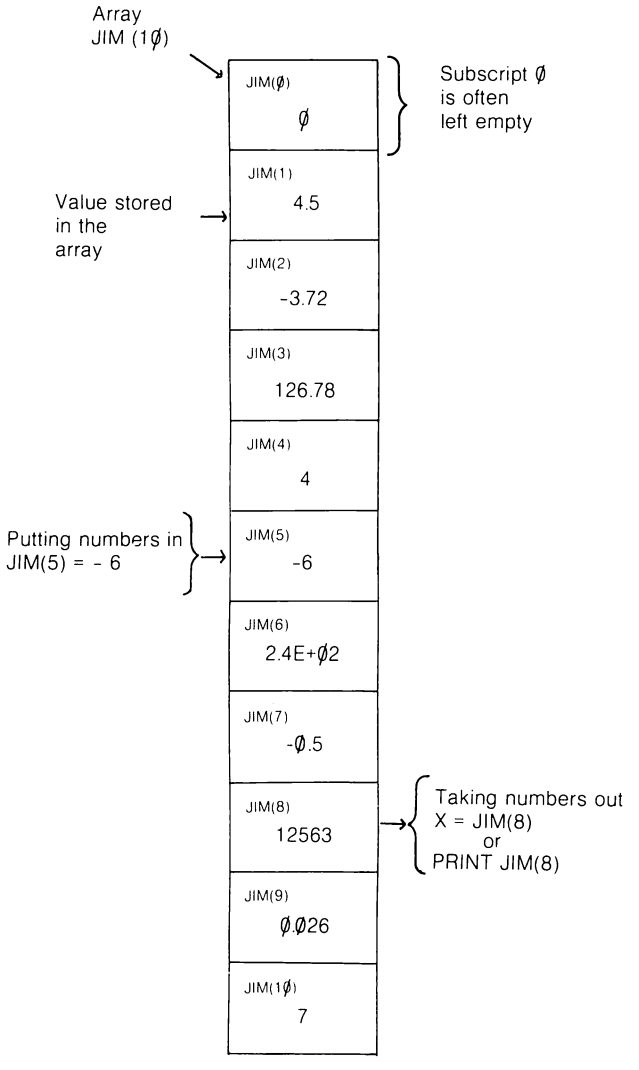


Fig. 5.2. A one-dimensional array called JIM. Because it is a numeric array, it can hold all kinds of numbers.

The second half of the program takes the numbers from JIM() one by one and prints them out on the screen. RUN the program. You are asked to key in 1Ø numbers, one after the other. They are each stored as a list, starting at JIM(1) and finishing at JIM(1Ø). As soon as you have finished keying in, the program goes on to print out all the numbers.

Now change the second half of the program:

```
5Ø INPUT"WHICH SUBSCRIPT";X
6Ø PRINT JIM(X)
7Ø GOTO 5Ø
```

RUN this program. To start with, you key in 1Ø numbers, as before. The second half now lets you find out what has been stored in any line of JIM(). When you are asked to type in a subscript number, key in any number between 1 and 1Ø. The computer then prints the number which is listed in JIM under that subscript number. If you type 'Ø', you get the reply 'Ø', for no value has been stored in JIM(Ø). If you type '11' or some other number greater than 1Ø, you see the reply '?BAD SUBSCRIPT ERROR'. The array has been dimensioned with 10 subscripts and this cannot be exceeded.

This program demonstrates one of the advantages of arrays. You can get any number out of it simply by quoting the subscript. For example, if you want the fourth number in the array, just use the term JIM(4). There is no need to work your way through the whole list, as with a DATA statement.

Another way of putting data into an array is to transfer it from a DATA statement:

```
1Ø DIM A$(5)
2Ø FOR J = 1 TO 5
3Ø READ A$(J)
4Ø NEXT J
5Ø FOR J = 5 TO 1 STEP -1
6Ø PRINT A$(J);
7Ø NEXT J
8Ø END
1ØØ DATA "Y", "A", "R", "R", "A"
```

This uses a string array, A$(), dimensioned in line 1Ø (actually there is no *need* to dimension an array if it is for less than 11 subscripts, including zero, as any undimensioned array is automatically given the dimension 10). Since A$() is a string array the DATA line must contain strings, which is the reason for the quotation marks. Lines 2Ø to 4Ø fill A$() with the data from line 1ØØ. Then lines 5Ø to 7Ø print it out, but in reverse order, to form an appropriate word. Note the use of the semicolon (;) after 'PRINT A$(J)'. The effect of this is to prevent the computer from beginning a new line when it prints the next string. All the strings are printed on the same line, one

following directly after the other, so building up the word from single letters.

The advantage of transferring data from a DATA statement to an array is that it is then much easier to handle. We can pick out any item simply by using its subscript. Also, we can perform many other kinds of operation that are not possible with a DATA statement, as in this program, where we print in reverse order.

The next program, Carousel, uses an array with two dimensions. We can think of this not as list but as a table, with rows and columns. There are two sets of subscripts, one for the rows, the other for the columns. The array A() used in Carousel has 8 rows and 3 columns (Fig. 5.3), so it is dimensioned by the statement 'DIM A(8,3). We leave the discussion of arrays now, to go on to discover the delights of the Carousel.

Carousel

This is a program to play music in the manner of a fairground

```
10 DIM N(12),A(8,3)
20 S1=36876:S2=36875:S3=36874:S4=36877:V
=36878
30 FOR J = 1 TO 12
40 PRINT"NO.";J;:INPUTN(J)
50 NEXT J
60 FOR J = 1 TO 8
70 FOR K = 1 TO 3
80 READ A(J,K)
90 NEXT K
100 NEXT J
110 FOR J = 1 TO 12
120 POKE S2,A(N(J),2):POKES4,253
130 POKE V,15
140 GOSUB 1000
150 GOSUB 1000
160 POKE S1,A(N(J),1):POKE S3,A(N(J),3)
170 GOSUB 1000
180 POKE S1,0:POKE S3,0
190 GOSUB 1500
200 POKE S1,A(N(J),1):POKE S3,A(N(J),3)
210 GOSUB 1000
220 POKE S1,0:POKE S3,0
230 IF J/3 = INT(J/3) AND F=0 THEN F=1:G
OTO 140
240 F=0
250 POKE S1,0:POKE S4,0
260 GOSUB 1500
270 NEXT J
280 GOTO 110
1000 FOR K=1 TO 100:NEXT K:RETURN
1500 FOR K=1 TO 30:NEXT K:RETURN
2000 DATA 230,193,230,232,200,232,214,20
6,222,218,209,230
2010 DATA 232,214,227,230,218,232,214,22
2,232,218,224,214
```

Listing 5.6. The Carousel.

carousel. It makes use of two arrays, N() and A(). N() holds the tune which is to be played. The tune is held as a list of twelve numbers. Each number has a value corresponding to the eight notes of the scale from middle C to top C. A() holds the numbers which have to be POKEd to the tone generators to produce the tones we want.

The program beings by dimensioning the arrays (line 1Ø). You need type DIM only once, followed by a list of arrays, separated by commas. N() is a list with 12 subscripts (ignoring N(Ø)), while A is as shown in Fig. 5.3. Line 2Ø defines some variables which are the addresses of the three tone generators (S1,S2,S3), the noise generator (S4) and the volume control (V).

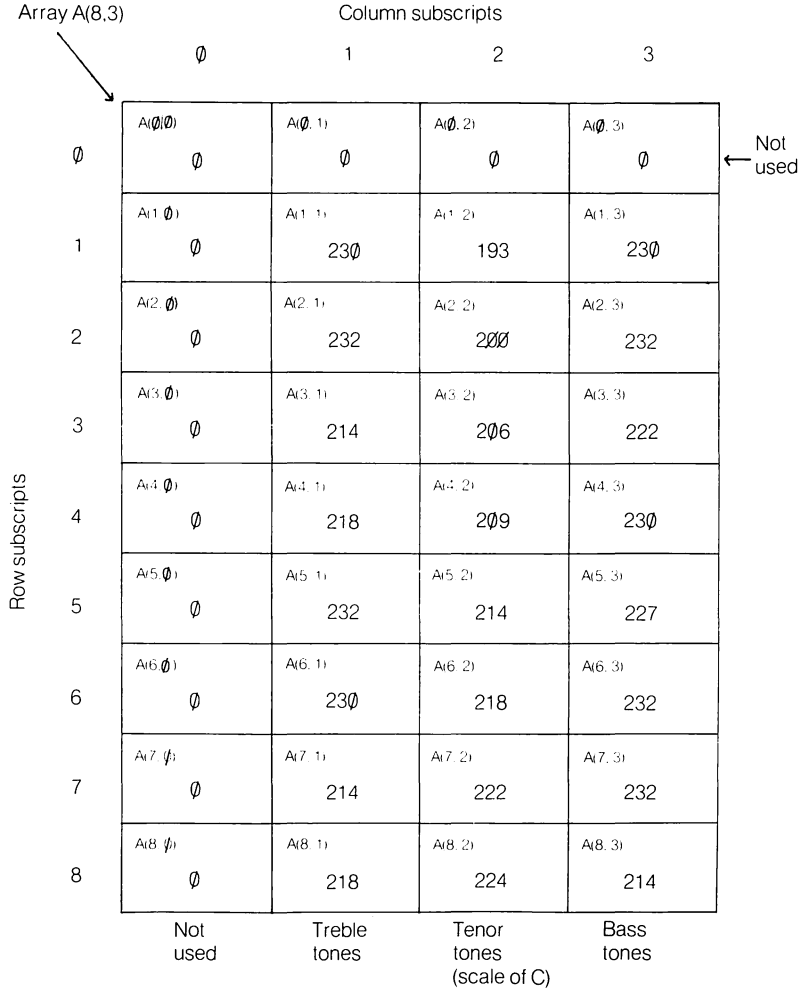


Fig. 5.3. A two-dimensional array holding values transferred from the DATA statements of the Carousel program.

When the program is RUN, the user is asked to key in the tune (lines 3Ø to 5Ø) to be put in N(). Key in twelve numbers, all between 1 and 8. These are the notes the numbers stand for:

Number	1	2	3	4	5	6	7	8
Note	C	D	E	F	G	A	B	C'

Do not worry about trying to key in any particular tune. The carousel makes even a straight run up the scale into an attractive melody.

As soon as the twelfth number has been keyed in, the program fills A() with the frequency numbers from the DATA statements (lines 6Ø to 1ØØ). This is done by using nested loops. The outer loop increments J from 1 to 8, to fill each row in turn. For each row, the inner loop increments K from 1 to 3 to fill each column in turn. Thus the array is filled in, just as if you were copying the figures from lines 2ØØØ and 2Ø1Ø row by row, working from left to right along each row in turn from top to bottom.

Now the music begins! There is a large loop (lines 11Ø to 27Ø) which repeats once for each note of the tune. First of all the Tenor tone generator is set to the note specified by the middle column of the array. Figure 5.4 shows how the expression A(N(J),2) is interpreted by the computer. Volume is set to its maximum (line 13Ø), for all Carousels play as loudly as possible. The Tenor tone generator always dominates the other two, making the melody sound clearly above the accompaniment.

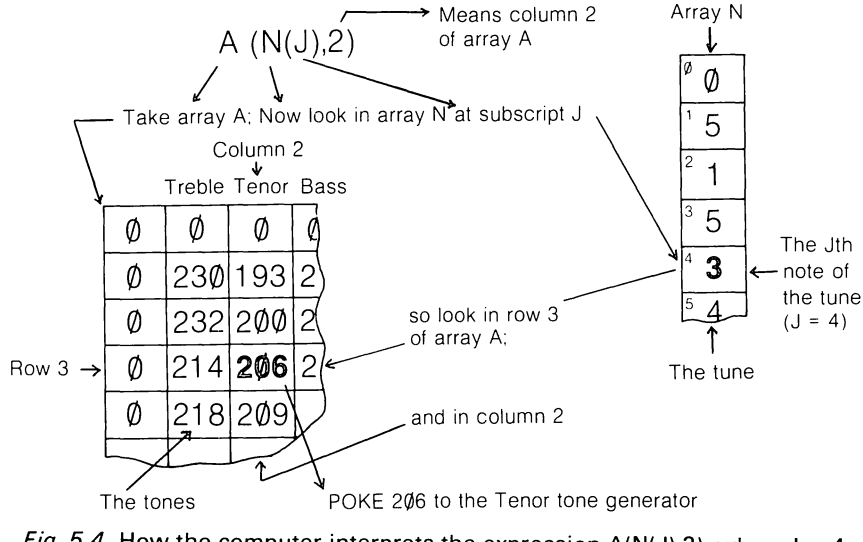


Fig. 5.4. How the computer interprets the expression A(N(J),2), when J = 4.

The second command in line 12Ø sets the noise generator to produce the effect of steam which we expect to hear when listening to a Carousel organ.

The note is held by using a subroutine (line 1ØØØ), which holds a delay loop. We get from the main program to this subroutine by the statement 'GOSUB 1ØØØ' at line 14Ø. When the delay is over, the command 'RETURN' at the end of line 1ØØØ sends the computer back to the line following the one that it came from. This means that it goes back to line 15Ø. Here it finds 'GOSUB 1ØØØ' again, so off it goes for another delay, then RETURNs to line 16Ø. We have used the subroutine twice to give a delay of double length.

The tenor note is still sounding, and now we add the sounds of the other two tone generators (line 16Ø). These too are POKEd with values taken from array A. The values come from the same row as that used to POKE the Tenor note, but the Treble note comes from column 1 and the Bass note from column 3. The values have been selected to produce notes which harmonise with the Tenor melody. The accompaniment is turned on for a single delay period (line 17Ø), off for a shorter period (subroutine 15ØØ, from line 19Ø), then on again for a single period (line 21Ø) and finally off (line 22Ø).

What happens next depends on the exact value of J. Every third Tenor note is to be sounded for twice the normal period (Fig. 5.5). Line 23Ø shows how it is possible to tell when this is to happen. The Tenor note is to be extended when J equals 3, 6, 9, and 12. All these values give *no remainder* when divided by 3. The result is an *integer*. So, if we divide J by 3 and then use the INTeger function on the result it makes no difference to its value.

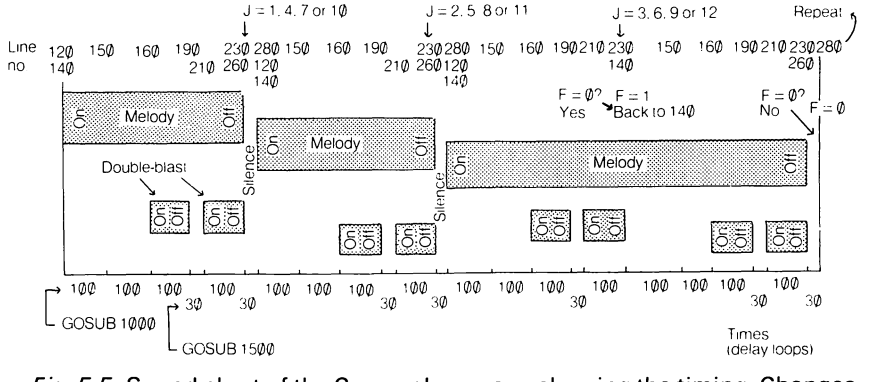


Fig. 5.5. Sound chart of the Carousel program, showing the timing. Changes of frequency depend on the notes selected for the tune. Volume is 15, except when there is silence.

For example,	$6/3 = 2$	and	$\text{INT}(6/3) = 2$,
but	$7/3 = 2.3333$	and	$\text{INT}(7/3) = 2$.

For numbers divisible by 3, the statement in line 23Ø is true and the micro is sent back to line 14Ø to repeat the double-blast of the Bass and Treble tones. But things are not quite as simple as one might think. When the computer gets back to line 23Ø for the second time, how is it to know that it is not to go back yet again to line 14Ø? The secret is the variable F. It has been given this name to remind us that it stands for 'flag'.

A *flag* is a variable set to (usually) either 0 or 1 to indicate to the micro that something has or has not happened. In this program, F is set to 1 when the micro has been sent back to line 14Ø to repeat the double-blast (Fig. 5.6). You can see this in line 23Ø.

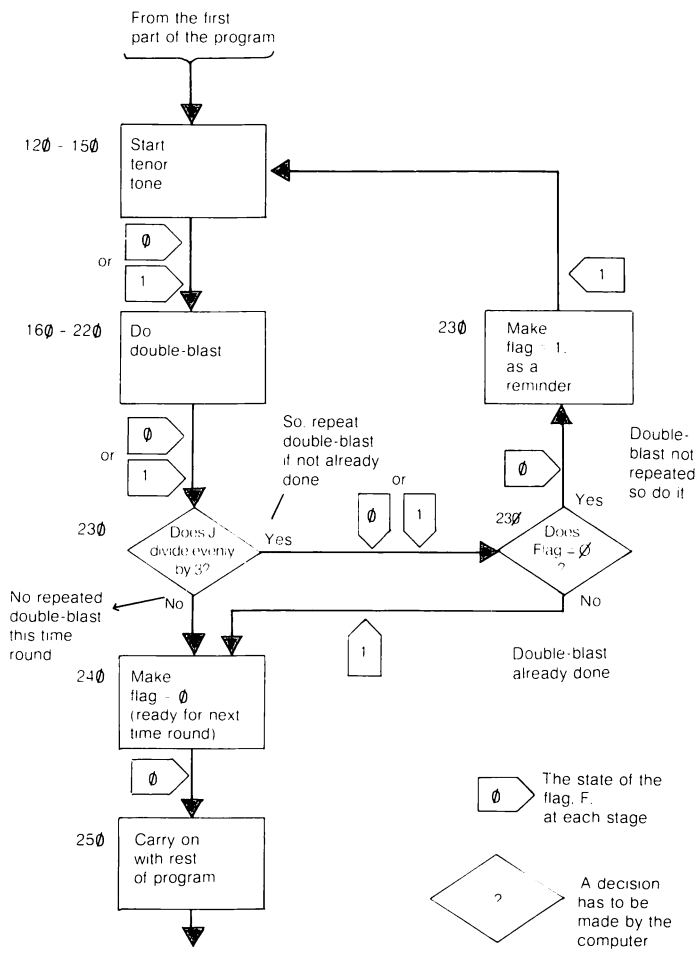


Fig. 5.6. How the flag works. The numbers are line numbers in the Carousel program.

If the micro gets to line 23ϕ and J is *not* divisible by 3, it carries straight on to the end of the loop, silencing the Tenor tone and steam at line 25ϕ. After a short pause (line 26ϕ) it goes back to line 11ϕ and then plays the next note.

If, at line 23ϕ, J *is* divisible by 3 *and* the flag is zero, it goes back to line 14ϕ to extend the note and repeat the double-blast. But, as it leaves line 23ϕ, F is set to 1.

When it gets back to line 23ϕ, J is still divisible by 3, but now F = 1. So the statement on line 23ϕ is not completely true. 'Not completely true' is the same as 'untrue' so the micro continues to line 24ϕ. Here F is reset to zero, ready for the next time that J is divisible by 3. This is a typical use of *flag variable*. We will see some more examples in later programs.

After playing the whole tune, the sequence is repeated indefinitely because of the 'GOTO 11ϕ' in line 28ϕ.

Colour too

The Carousel plays so prettily that it is a pity not to have something pleasing to look at on the screen. Type in these extra lines:

```
55 PRINT"■" :POKE36879,140:PRINT:FORJ=1TO
11:PRINT"■■■X";:NEXTJ
56 PRINT:PRINT:PRINTTAB(7)"■■■
```

```
10 DIM N(12),A(8,3)
20 S1=36876:S2=36875:S3=36874:S4=36877:V
=36878
30 FOR J = 1 TO 12
40 PRINT"NO.";J;:INPUTN(J)
50 NEXT J
55 PRINT"■":POKE 36879,140:PRINT:FOR J=1
 TO 11:PRINT"■■■■";:NEXT J
56 PRINT:PRINT:PRINTTAB(7)"■■■■■■■■■■■■
E■■■"
60 FOR J = 1 TO 8
70 FOR K = 1 TO 3
80 READ A(J,K)
90 NEXT K
100 NEXT J
110 FOR J = 1 TO 12
120 POKE S2,A(N(J),2):POKES4,253
130 POKE V,15
135 POKE 36879,26+J*16+INT(RND(1)*6)
140 GOSUB 1000
150 GOSUB 1000
160 POKE S1,A(N(J),1):POKE S3,A(N(J),3)
170 GOSUB 1000
180 POKE S1,0:POKE S3,0
190 GOSUB 1500
200 POKE S1,A(N(J),1):POKE S3,A(N(J),3)
210 GOSUB 1000
220 POKE S1,0:POKE S3,0
230 IF J/3=INT(J/3) AND F=0 THEN F=1:GOT
O 140
240 F = 0
250 POKE S1,0:POKE S4,0
260 GOSUB 1500
270 NEXT J
280 GOTO 110
1000 FOR K=1 TO 100:NEXT K:RETURN
1500 FOR K=1 TO 30:NEXT K:RETURN
2000 DATA 230,193,230,232,200,232,214,20
6,222,218,209,230
2010 DATA 232,214,227,230,218,232,214,22
2,232,218,224,214
```

Listing 5.7. Adding colour to the Carousel.

a different colour. Before typing each letter, set its colour by pressing CTRL with one of the number keys.

Line 135 changes the screen and border colours at random, in time with the music. The rather strange expression used is one that avoids having the screen or border printed black or white. We want nothing as dreary as black or white when the Carousel begins to play!

To summarise

In this chapter, you have found out how to:

- use the VIC's sound generators and the volume control
- use INPUT to accept two variables at one time
- store information in DATA statements and READ it from the DATA statements for use in a program

- use the RESUME command to make the computer READ from the beginning of the data statements
- set out nesting FOR...NEXT loops
- dimension arrays, how to put data into arrays, and how to read data from them
- use a variable as a flag
- make the VIC print letters in different colours.

Chapter Six

The Organised Machine

We will take a short rest from the sounds of music and the gaudy colours of the Carousel to look at some of the ways in which we make the VIC do exactly what we want. You have already used many of the commands which are to be described in this chapter, but their scope will be explained in fuller detail. There are also several new BASIC keywords which will be used in later chapters.

Repeating

Micros are very good at repeating themselves. Unlike most people, they do not become bored by having to do the same thing over and over again. To see this in action type the program shown in Listing 6.1.

```
10 FOR J = 1 TO 10000
20 PRINT J;"BORING!"
30 NEXT J
```

Listing 6.1. The most boring program ever written (maybe!).

RUN this program. If you get bored before the micro does, you only have to press RUN/STOP. Good! Having made the point, we can get on with the chapter.

Actually, the computer was not doing exactly the same thing each time round the loop, for the value of J was being incremented. Each time round, it printed a different number, but counting from 1 to 10000 is *still* a rather boring task. The program in Listing 6.2 uses the FOR...NEXT command to do something more useful.

```
10 PRINT"□"
20 INPUT"WHICH TABLE";N
30 FOR J = 1 TO 10
40 PRINT J;"TIMES";N;"IS";J*N
50 NEXT J
```

Listing 6.2. Quick multiplication tables.

When you RUN this program, you are asked "WHICH TABLE?". The micro is going to print out a multiplication table, and wants to know which table you want. Key in a number (any number, even a negative one) and press RETURN. The table is then printed out. Each time around the loop J is incremented by 1 and a line of the multiplication table is printed. Now try altering line 3Ø of the program to:

```
3Ø FOR J = 1 TO 1Ø STEP 2
```

The STEP command causes J to be incremented by 2 each time round, instead of by 1. Now the table consists only of the lines for '1 TIMES', '3 TIMES', '5 TIMES', '7 TIMES', and '9 TIMES'. After this, J is incremented to 11 but, since this is greater than the largest value allowed by the command 'J = 1 to 1Ø', the loop is not repeated again. What should line 3Ø be to make the micro print out the lines for '2 TIMES', '4 TIMES', ..., '10 TIMES'? Type in what you think the line should be and see if it works. Now try to get the micro to print the lines for '1 TIMES', '4 TIMES', '7 TIMES' and '10 TIMES'.

To make the VIC say its tables backwards, type:

```
3Ø FOR J = 1Ø TO 1 STEP -1
```

What happens if you leave out the 'STEP −1'? The micro prints the line for '10 TIMES' and then stops. This shows that it will always go through the loop *once*, even if the instructions make nonsense.

There is no need to make the STEP a whole number. Try this:

```
3Ø FOR J = 1 TO 2 STEP .1
```

or this:

```
3Ø FOR J = -4 TO 4 STEP .5
```

There is no end to the multiplication tables that can be compiled. Finally, try this one:

```
3Ø FOR J = 1E2 TO 1E3
```

This shows that exponentials are quite acceptable in a FOR...NEXT loop. It also demonstrates that humans becomes bored more quickly than micros. Once again, the RUN/STOP key proves its usefulness.

As mentioned in Chapter Five, FOR...NEXT loops may be nested one within the other. You have to use a different variable as the 'counter' variable for each loop, as in the program in Listing 6.3, where the counters are J in the outer loop and K in the inner loop.

```
10 PRINT"□"
20 INPUT"WHICH TABLES";N,M
30 FOR J = 1 TO N
40 FOR K = 1 TO M
50 PRINT J*K;
60 NEXT K
70 PRINT""
80 NEXT J
```

inner loop

outer loop

Listing 6.3. Even more multiplication tables.

This program prints several tables at the same time, as you will see when you RUN it. This time you must key in two numbers. It is better if K is not greater than 6, for otherwise each table takes more than one line and the display becomes confused.

For each value of J, the inner loop prints a set of values of J times K, the 'J times table' from 1 to K. The PRINT statement in line 5ϕ ends with a semicolon. This makes the computer print all the answers on the same line of the screen. When it has printed out one table, it comes to the end of the inner loop and goes on to line 7ϕ. Here it is made to print a null string (i.e. nothing). Note that this line does not end with a semicolon. This line is there simply to make the computer go on to the next *screen* line to print the next table.

It is possible to nest more than two loops one within the other. For example, you could have the program shown in Listing 6.4.

```
10 FOR J = 1 TO 10
20 FOR K = 1 TO 100
30 FOR L = 1 TO 1000
40 FOR M = 1 TO 10000
50 PRINT"MOST BORING!"
60 NEXT M
70 NEXT L
80 NEXT K
90 NEXT J
```

Listing 6.4. Nested loops, resulting in a program even more boring than that in Listing 6.1.

These loops each nest completely inside the next one to the outside. We start with J, K, L and M, and leave in the reverse order: M, L, K, J. After printing the message ten thousand million times, the micro takes a welcome rest.

As we have seen, the values given in a FOR...NEXT loop can be constants (FOR J = 1 TO 1ϕ), variables (FOR J = N TO M) or expressions (FOR J = 2*M/3 TO 4*(N+1)+23). The thing to watch out for is that none of these values may be changed *during the running* of the loops. For example, if the values of N or M are re-calculated *within* the loop, this can make the program crash. If you want to recalculate values while a loop is running, use the method shown in the example in Listing 6.5.

The flow chart (Fig. 6.1) explains how the program works. It

```
10 PRINT"□"
20 INPUT"MAXIMUM";M
30 N = 0
40 J = 1
50 N = N + J
60 PRINTN
70 IF N < M THEN J = J + 1: GOTO 50
80 END
```

The loop

Listing 6.5. Constructing a loop to allow values to be re-calculated (the equivalent of REPEAT ... UNTIL).

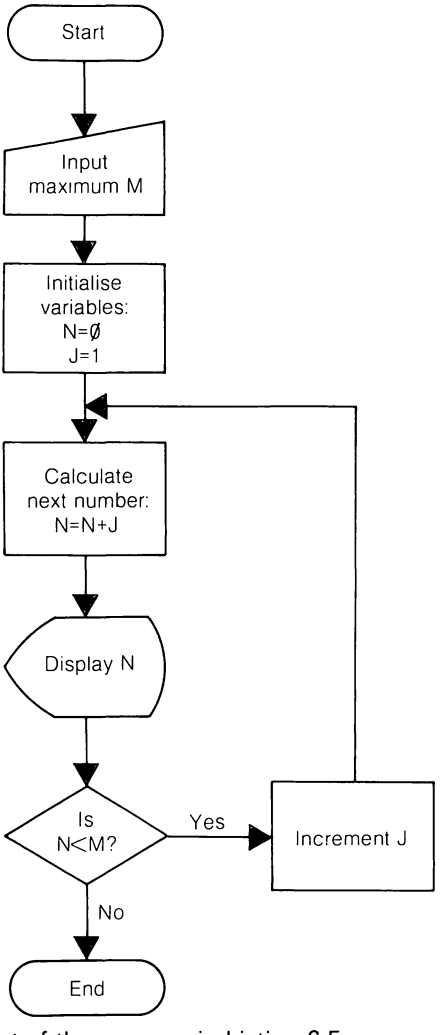


Fig. 6.1. Flow chart of the program in Listing 6.5.

prints out a series of numbers: 1, 3, 6, 10, 15, 21, etc. The differences between successive numbers increase by 1:

The numbers:	1	3	6	10	15	21
The differences:		2	3	4	5	6

At the beginning of the program you are asked to set a maximum value. J is the loop counter. It is not easy to work out in advance how many times the loop must be repeated. This means that we cannot use an ordinary FOR...NEXT loop. Instead the program calculates and prints each number and then finds out if it has gone far enough. If not, it goes back and prints the next number. We have not used a special command such as FOR...NEXT, but have constructed the loop from other commands.

One new symbol appears in this program, in line 7Ø. 'IF N <M' means 'IF N is less than M'. In other words, as long as the numbers being printed are less than the maximum you have set, the loop repeats. When N equals N or becomes greater than M, this final N is printed but the loop is not repeated and the program ends.

Timing

Quite often, we use a FOR...NEXT loop as a delaying tactic. There have been several instances in programs earlier in the book where we have made the computer run through a loop such as 'FOR K = 1 TO 3ØØØ: NEXT K' in order to prevent it from rushing on to carry out the next command. This is a useful way of obtaining delays of various lengths. Counting to 1000 takes approximately 1 second, so the '1 TO 3000' loop gives a delay of roughly 3 seconds. The Air Raid program in Listing 5.4 showed how to obtain a delay of random length.

The special variable TI is used to obtain a delay of accurately known length. In a games program, for example, it might be necessary to allow each player a fixed amount of time in which to make a move. The program in Listing 6.6 gives a player two minutes in which to make a move, and then prints "TIME UP":

```
10 S = 120
20 TI$ ="000000"
30 IF TI < S * 60 THEN GOTO 30
40 PRINT"TIME UP"
```

Listing 6.6. An elapsed-time program.

The variable S is the length of time required, in seconds. At line 2Ø, the VIC's clock is reset. Making TI$ equal to 'all zeros' resets TI too. TI is incremented 60 times a second. The micro repeats line 3Ø until TI equals or exceeds 7200, then prints the message. A routine such as this is best used as a subroutine. To convert this program to give *any* required time delay, change line 1Ø to:

```
1Ø INPUT"HOW MANY SECONDS";S
```

Now you will never be without a high-precision egg-timer!

Taking decisions

When the computer comes across a line of the form:

IF (condition) THEN (action)

it has to decide whether the condition is true or false. The condition could be a simple equality such as:

60 IF X = 10 THEN GOTO 120

If the condition (X = 10) is true, that is to say X really does equal 10, the micro takes the specified action. In this example it would jump from line 60 to line 120, and continue the program from there. But if the condition is not true (X has some value other than 10) the action is not performed. The micro does nothing, and drops down to the next line after line 60.

The 'equals' sign in the condition above is one of several *relational operators.* It specifies a relationship between two quantities. In this example, the quantities are 10 and the value of X. There are two other relational operators:

< which means 'less than'
> which means 'greater than'

We can also combine the relational operators in pairs:

X <= 10 means 'X is less than or equal to 10'
X >= 10 means 'X is greater than or equal to 10'
X <> 10 means 'X is less than or greater than 10'

The last one is the same as saying 'X is not equal to 10'. There is a program in Listing 6.7 to demonstrate how these work.

The program in Listing 6.7 is called 'Agequiz'. You type in your age and the computer makes one or more appropriate responses. For example, if your age is 17, the conditions at lines 40, 50, 60 and 70 are false. The computer drops through all these lines without doing anything. When it reaches line 80 it finds a condition which is true - your age is not equal to 21. It carries on along line 80 and prints the message "WOULDN'T YOU LIKE TO BE 21?".

The condition at line 90 is true too, for your age is greater than 12 and it is also less than 20. So it prints "HELLO, TEENAGER!".

```
10 PRINT"□"
20 INPUT"WHAT IS YOUR AGE";A
30 PRINT
40 IF A < 0 THEN PRINT"DON'T BE SILLY!"
50 IF A <= 5 THEN PRINT"YOU ARE TOO YOUN
G TO  HAVE READ THIS"
60 IF A = 5 THEN PRINT"STARTING SCHOOL S
OON?"
70 IF A > 5 AND A < 17 THEN PRINT"HAVING
 FUN AT SCHOOL?"
80 IF A <> 21 THEN PRINT"WOULDN'T YOU LI
KE TO  BE 21?"
90 IF A > 12 AND A < 20 THEN PRINT"HELLO
, TEENAGER!"
100 IF A => 20 THEN PRINT"WHAT IS IT LIK
E TO BE";A;"?"
110 IF A => 80 THEN PRINT"CONGRATULATION
S!"
120 IF A > 100 THEN PRINT"???"
130 END
```

Listing 6.7. Agequiz, a program which shows how to use relational operators.

From then on the micro finds no more true conditions, and prints nothing more.

The comments given in the program are rather general ones, but you could have fun by replacing them with comments which fit members of your family.

The conditions in lines 7Ø and 9Ø include the BASIC keyword AND. This word is acted on just as it is understood in an ordinary English sentence. The condition is true only if both parts of it are true. If you are 17, your age is greater than 5 but it is not less than 17. Only one of the conditions is true at line 7Ø, so the statement as a whole is false. At line 9Ø, on the other hand, your age is greater than 12 AND it is less than 20, so the statement as a whole is true and you are greeted as a teenager.

Although AND appears as an ordinary word it is much more significant than that. It is one of three *logical operators* which the computer understands. The other two are OR and NOT. With OR we need two conditions but only one of them need be true. You could add this line:

```
95 IF A <13 OR A>19 THEN PRINT "YOU ARE NOT A
TEENAGER"
```

If you are aged 11, the first condition is true, the second is false. But OR requires only one OR the other to be true, so the statement as a whole is true and it prints the message. If you are aged 35, the first condition is false, but the second is true, so the message is printed. However, if you are a teenager, both conditions are false, making the whole statement false, and the message is not printed.

The NOT operator makes a condition false if (without the NOT) it

is true. For example, you could alter line 40 to:

40 IF NOT A => 0 THEN PRINT "DON'T BE SILLY!"

This has the same effect as the existing line, so there is little point in doing this. However, there are occasions when using NOT makes the logic easier to follow – perhaps this line makes more sense to *you* in this form.

You can combine several ANDs, ORs and NOTs in statements which include several conditions. The 'Animals' program in Listing 6.8 illustrates how we do this.

```
10 PRINT"□"
20 PRINT"WHAT IS IT?":PRINT
30 INPUT"HOW MANY LEGS";L
40 INPUT"HOW MANY WINGS";W
50 INPUT"DOES IT FLY (Y/N)";F$
60 IF F$="Y" THEN F = 1
70 IF L = 2 AND W = 0 AND F = 0 THEN A$=
"MONKEY"
80 IF L = 2 AND W = 2 AND F = 0 THEN A$
= "KIWI"
90 IF L = 4 AND W = 0 AND F = 0 THEN A$
= "COW"
100 IF L > 20 AND W = 0 AND F = 0 THEN A
$ = "MILLIPEDE"
110 IF (L = 4 OR L = 2) AND W = 2 AND F
= 1 THEN A$ = "BAT"
120 IFL=0ANDW=0ANDF=0 OR L>5ANDL<20ANDW=
0ANDF=0 OR L=6ANDW=4ANDF=1 THEN A$="MOTH
"
130 PRINT"□"
140 IF A$="" THEN GOTO 170
150 PRINT"IT MIGHT BE A ";A$;"."
160 END
170 PRINT"I DO NOT KNOW THIS     ANIMAL."
180 END
```

Listing 6.8. The Animals program shows how to use logical operators.

The idea of the program is simple. You think of an animal and then answer questions about it. There are only three questions (see lines 30 to 50). The computer than tries to work out what animal you are thinking of. Obviously, you will not get much response from the computer unless you think of one of the animals it knows something about (see lines 70 to 120).

You tell the computer the number of legs (L) and wings (W) and if the animal can fly. If it can fly, F = 1; if it cannot fly, F= 0. In lines 70 to 120 the micro performs logical operations on L, W and F to discover the name of the animal. Line 70 tells it that if it has 2 legs AND has no wings AND cannot fly, it is a monkey. Here we use two ANDs, so all three conditions must be true if the computer is to print out "MONKEY". Similar logic applies in lines 80 and 90. Of course, it is possible to simplify lines 70 to 90 by cutting out the 'AND F = 0' from each, but this might make difficulties if you decided to add

other animals to the program, for example, a sparrow.

Line 1ØØ allows users to enter any number greater than 2Ø for the legs of a millipede.

Some people would say that a bat has 2 legs, but others might say it has 4 - it all depends what you mean by 'leg'. Line 11Ø lets you enter either 2 or 4 as the number of legs. Notice how this line is set out. You might think that it would be correct to write the line like this:

```
11Ø IF L = 4 OR L = 2 AND W = 2 AND F = 1 ...
```

This does not give the same result because the computer works out all the ANDs before it works out the OR. In other words there is a rule of priority, just as there was for the arithmetic operators (see Chapter Four). Without the brackets, the statement is true if:

```
    L = 4 (no matter what the values of W and F)
OR  L = 2 AND W = 2 AND E = 1
```

So any four-legged animal (e.g. cow) would result in the answer 'BAT'. The rules of priorities cause the micro to operate on expressions inside brackets first. The brackets in line 11Ø force the micro to operate on the OR first. If L = 4 OR L = 2, the expression in the brackets is true (bats have 2 or 4 'legs'). Then if this is true AND W = 2 AND F = 1, the animal in question is a bat.

Line 12Ø caters for the complex life story of the moth. As an adult moth it has 6 legs and 4 wings and can fly. But as a caterpillar it has 6 true legs and some other leg-like appendages, no wings and cannot fly. As a pupa (or chrysalis) it has no legs, no wings and cannot fly. The micro has to consider the three possible combinations of features:

```
    No legs, no wings, cannot fly (moth pupa)
OR  Between 6 and 19 'legs' (depending on what you take to be a
    leg), no wings and cannot fly (moth caterpillar)
OR  6 legs, 4 wings, can fly (adult moth)
```

If the features for any one of these are correct, it is a moth. The various groups of features are ANDed together to see if the details typed in fit any one of the stages in the moth's life story. If so, it is a moth. Note that since AND has priority over OR there is no need for brackets in line 12Ø.

The computer is able to organise its logic in another way, which you may find easier to use in programming. The program in Listing 6.8 above asked "*How many* legs?", and "*How many* wings?" and

then worked with numbers of different values. We can simplify the logic if we ask questions which have only *two* possible answers – "Yes" or "No", or perhaps "True" or "False". The corresponding variables are given the value '0' to correspond with "No" (or "False") or the value '−1' to correspond with "Yes" (or "True"). The computer understands this system so, for example, if we make A = −1 and B = −1 then at the line:

10 IF A AND B THEN PRINT "TRUE"

it prints "TRUE". The condition is true only if both A *and* B are true (both are '−1'). But if A and B have any other pairs of values (0,0 or 0, −1, or −1,0), then A AND B is false, so the message is not printed. Listing 6.9 is a similar animal guessing program, 'Animalogic', which shows how this type of logic can be used. It certainly makes the logic of the program easier to follow.

```
10 PRINT"□"
20 PRINT"WHAT IS IT?"
30 INPUT"HAS IT GOT WINGS";W$
40 IF W$="Y" THEN W=-1
50 INPUT"CAN IT FLY";F$
60 IF F$="Y" THEN F=-1
70 INPUT"HAS IT A FURRY COAT";C$
80 IF C$="Y" THEN C=-1
90 IF W AND F AND C THEN A$="BAT"
100 IF NOT W AND F AND NOT C THEN A$="MO
NEY-  SPIDER"
110 IF W AND NOT F AND NOT C THEN A$="OS
TRICH"
120 IF NOT W AND NOT F AND NOT C OR W AN
D F AND NOT C THEN A$="MOTH"
130 PRINT"□"
140 IFA$="" THEN GOTO 170
150 PRINT"IT MIGHT BE A ";A$
160 END
170 PRINT"I DO NOT KNOW THIS ANIMAL."
180 END
```

Listing 6.9. Animalogic has another approach to logical programming.

In this program, all the questions requires Yes/No answers, so type either 'Y' or 'N' for each. At line 90, the statement as a whole is true if all three of W, F, and C are true, that is to say, if they all have value −1. Line 100 is to catch out people who like to try to fool the computer. The money-spider does not have wings, but it can fly for miles, attached to a thin thread of gossamer! It does not have a furry coat so C is false or, putting it another way, NOT C is true. Line 110 is an animal which has wings (W is true), cannot fly (NOT F is true), and does not have a furry coat (NOT C is true). Finally, on line 120 we have the more complicated logic of the moth. First we deal with the caterpillar and pupa together (no wings, no flying, no fur). After

the OR, we deal with the adult (wings, can fly, no fur). The computer processes the ANDs first, on both sides of the OR. Then, if one OR the other of these ANDed conditions fits the moth's features, it puts A$ equal to "MOTH".

Decisions by numbers

The BASIC keyword 'ON' provides us with a kind of 'signpost' pointing to many different parts of the program. The format of the command is:

ON N GOTO ...
or ON N GOSUB ...

You can put any other numeric variable, an integer variable or an arithmetic expression instead of N. The command is followed by a list of line numbers separated by commas. The program in Listing 6.10 shows how it works for GOTO.

```
10 PRINT"□"
20 INPUT"YOUR TURN";X
30 N = INT (RND(1) * 6) +1:PRINT
40 ON N GOTO 100, 200, 300, 400, 500, 60
0
100 PRINT"YOU LOSE 1"
110 GOSUB 1000
120 GOTO 10
200 PRINT"■YOU WIN 1"
210 GOSUB 1000
220 PRINT"■":GOTO 10
300 PRINT"■YOU LOSE 3 - BAD LUCK"
310 GOSUB 1000
320 PRINT"■":GOTO 10
400 PRINT"■YOU WIN 3 - HOORAY!"
410 POKE 36879,90
420 GOSUB 1000
430 PRINT"□":GOTO 10
500 PRINT"■PAY THE KITTY 2"
510 POKE 36879,24
520 GOSUB 1000
530 PRINT"□":GOTO 10
600 PRINT"■YOU ■WIN ■THE ■KITTY"
610 FOR K = 0 TO 6
620 POKE 36879,153 + K
630 GOSUB 1000
640 NEXT K
650 GOTO 10
1000 FOR J = 1 TO 2000: NEXT J
1010 RETURN
```

Listing 6.10. Put and Take, a gamble based on random numbers.

One point of interest in this program is the way it uses the RND function to produce a random number between 1 and 6, just as if you are throwing a dice. When you are ready, press the RETURN key. The value X is a dummy one which is not used in the program. The

random number is N and, on reaching line 4Ø, the computer GOes TO one of the lines listed there. Because it is 'ON N GOTO' it goes to the line number at position N in the list. For example, if N is 3, it goes to the third line listed, line 3ØØ. At line 3ØØ is the part of the program which tells you what to do when you have 'shaken' 3 on the 'dice'. Note that although this program has a neat list of lines in even hundreds, and in numerical order, this is just to make this program simpler to set out. However, in another program you could have the line numbers in any order and they need not be even hundreds:

```
1ØØ ON Z GOTO 123, 456, 321, 6, 777
```

In this example, with Z equal to 3 the micro jumps to line 321.

This program is a simple but engaging gambling game for two players. You could easily modify it to make it into a family game. There are plenty of line numbers to spare in each section of the program so there is scope for improving the visual displays. You could also add some sound effects. The FXG program in Listing 9.1 of Chapter Nine will help you do this.

To summarise

In this chapter, you have found out how to:

- program FOR...TO...STEP...NEXT loops
- nest loops properly
- make the program repeat until a certain condition is true
- use ON...GOTO and ON...GOSUB
- use the variable TI
- use the relational operators, = < and >
- write a branching program, using IF (condition) THEN (action)
- use the logical operators, AND, OR and NOT, either for relating two or more conditional statements, or for relating two or more variables which have logic values.

You have learned that:

- a variable with value 'Ø' has logic value "False"
- a variable with value '−1' has logic value "True"
- the rule of priority of logical operators is:
  First, expressions inside brackets
  Then, AND
  Then, OR
  Last, NOT.

Chapter Seven
More About Graphics

There are three ways of programming the VIC to give a colourful display on the screen:

(1) use a PRINT statement
(2) POKE the video RAM
(3) use the CHR$ function.

We have often used the first two methods, but the third is a new one. The third one is less often used, possibly because the other two methods are more convenient, but it as well to know about it.

All the letters, numerals, symbols and other characters used by the computer have a code number by which they are identified. This is known as the *ASCII code*. The letters stand for American Standard Code for Information Interchange. These codes were used originally for sending data between one teletypewriter and another. The CHR$ function uses ASCII codes in this way:

```
1Ø PRINT CHR$(65)
```

When you RUN this short program, the computer simply prints 'A' for, as shown in Table 10 (Appendix A), the ASCII code for 'A' is 65. This seems a rather long-winded way of printing 'A' when we could do it more simply with:

```
1Ø PRINT "A"
```

But there are some advantages in this method. It may happen that you want to be able to print one of a number of different characters but the one to be printed has to be decided *during* the running of the program. Then you could have a line such as:

```
25Ø PRINT CHR$(N + 6)
```

The character PRINTed depends on the value of N.

The ASCII code shows its ancestry when we consider the *control*

codes. An example is CHR$(13), which causes a 'carriage return'. It has the same effect as pressing the RETURN key. These control codes do not result in anything being printed on the screen, but affect the operation of the machine (teletypewriter or computer). Apart from CHR$(13), the control codes used by the VIC are not ASCII standard codes. They control the cursor, the colour of printing, and several other functions that are controllable from the keyboard. The graphics characters can also be printed using CHR$ but, since these characters are a feature found only on Commodore computers, their codes are not part of the ASCII standard.

Using POKE

We have had several examples of this method, which was the subject of Chapter Three. The numbers to be POKEd into the character code RAM (screen codes) are the same as the ASCII codes, *if* we are using numerals, symbols and capital letters in Text Mode. The lower-case letters are not standard ASCII codes and the codes for graphics are allocated in a complicated way. When using POKE, it is simpler to refer to Table 6 in Appendix A rather than to try to use ASCII codes.

Using PRINT

We have had a few examples of the use of PRINT (line 55 of the Carousel program, Listing 5.7). Now we will take this method further, for it is one of the best ways of building up pictures on the screen. The program in Listing 7.1, called Puffer is an example of what can be done.

```
10 PRINT"□":POKE 36879,156
20 N = 7
30 PRINT TAB(N)"    ■▒"
40 PRINT TAB(N)"     ▒"
50 PRINT TAB(N)"▀▄  ▄   ▄▀ "
60 PRINT TAB(N)" ▀▄      "
70 PRINT TAB(N)"▄        ▀"
80 PRINT TAB(N)"▀▄ ◠ ◠ "
90 PRINT TAB(N)"  ◡ ◡ "
```

Listing 7.1. Puffer.

It begins in the usual way by clearing the screen and sending the cursor back home. Remember that the reversed 'heart' PRINTed in line 1Ø does not appear as a heart on the screen. It was put into the print statement by pressing SHIFT with CLR/HOME. It appears in

the listing as a heart, but has no immediate effect when you type it in, or when you LIST the program. But when you RUN the program, it is interpreted as SHIFT with CLR/HOME. In other words, it is a *control character.*

Line 1Ø also changes the screen to light orange and the border to purple, to give a suitable background to the picture. The picture is PRINTed in lines 3Ø to 9Ø as a set of seven strings of characters. So that the picture will appear in the centre of the screen and not at the left-hand edge, we follow the command PRINT with the word TAB. This tells the computer to begin printing the strings at column N Line 2Ø has fixed the value of N as 7, so printing begins at column 7 instead of from column 0.

The strings contain both control characters (*not* to be printed) and screen characters (to be printed). For example, in line 3Ø the string begins with three spaces, followed by the control character for CTRL with '1'. This changes the printing colour to black. Immediately after this comes the shaded pattern (Commodore 'Flag' key with '+'), which represents the smoke coming from the stack of the Puffer. The next string, is similar but with the 'smoke' one column further to the right.

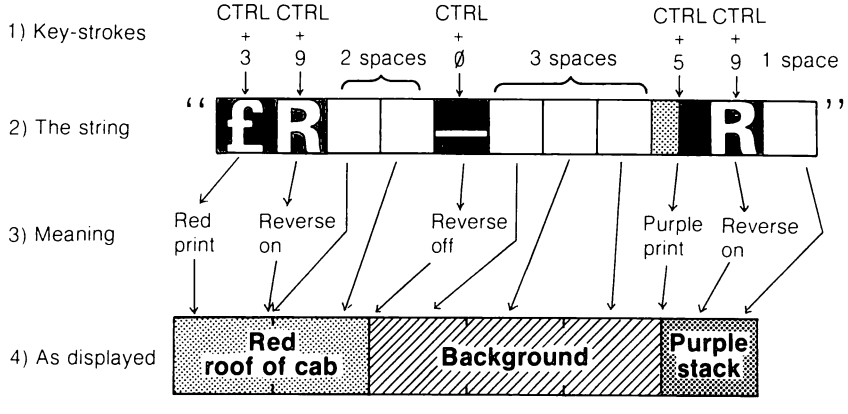


Fig. 7.1. Programming graphics by PRINTing strings (line 5Ø of Puffer).

Line 5Ø is more complicated. Figure 7.1 shows how it is made up. The reversed '£' means 'PRINT in red'. The reversed 'R' means 'reverse on' (obtained by pressing CTRL with 9). Then come two spaces. Because reverse is on, these are printed as solid red blocks, the roof of the cab. Following these we need some spaces which are not solid colour; they are to be the same colour as the screen. We have a control character for 'reverse off' followed by three spaces. Note how we use 'reverse on' to print solid blocks of colour, and

change to 'reverse off' to get the background colour. Following this, the colour is changed to purple. We command 'reverse on' and then print a space. This will later appear as a solid purple block, for the smoke-stack. When you run the program, you will be able to see the effects of the control characters used in the remaining lines.

One point to note is that a command to change colour remains in effect until it is *altered.* For example, in line 6Ø we are printing in red, so no further command for red is required for line 7Ø. In contrast, the command for 'reverse on' is cancelled automatically at the end of each PRINT statement. It needs to be renewed each time we want it. In line 7Ø, we need 'reverse on' to begin this string, even though we have used 'reverse on' in line 6Ø. The only time we need to use 'reverse off' is when the reversed state has to be cancelled *within* a string, as in line 5Ø, after printing the cabin roof.

How about putting the Puffer into motion? This is a rather crude attempt, but it illustrates some interesting points. Change line 2Ø of the 'Puffer' program to:

```
2Ø FOR N = Ø TO 14
```

This affects the value of N in the TAB commands, making the computer print the strings further to the right at each repetition.

Then add these lines:

```
1ØØ FOR J = 1 TO 15Ø:NEXT J
11Ø IF N <14 THEN PRINT" ♥ "
12Ø NEXT N
```

RUN the program and watch the Puffer move across the screen, coming to a halt on the right side. What the program does is to print the picture 15 times in quick succession, moving it across the screen toward the right, one column at a time. Line 1ØØ delays the loop long enough to allow you to see the picture in each position. Line 11Ø clears the screen ready for the next picture to be printed. The last time round, when N is 14, this line is skipped, so as to leave the picture of the 'Puffer' on the screen.

This illustrates the main features of animation. We print pictures on the screen in rapid succession. The pictures themselves are still, but because they differ slightly the objects in the picture appear to move. Sometimes, as in this example, we move the whole picture. Sometimes, as in some of the examples in Chapter Eleven, we move parts of the picture. More about this in Chapter Eleven! In the meantime, why not add some sound effects to the 'Puffer' program?

The character store

We have been using characters of all kinds without thinking about how the VIC knows to make the right shapes appear on the TV screen. Now we will look into this in more detail. Among its many 'chips', it has a special one known as a *character generator*. We can think of this as a set of boxes, each of which contains part of the instructions for forming a character. Several of these instructions have to be sent to the TV screen for every character that is displayed. We need not be concerned with exactly how the instructions are carried out by the TV set, but it is worth while finding out how the instructions are made up, so that you can design your own characters. If you want a Greek 'alpha' or any other letter of a foreign alphabet, for example, or a scientific or mathematical symbol, or perhaps an alien spacecraft, it is easy to convert some of the keys to produce such characters.

First of all, how are the characters stored? The best way to find this out is to look at a sample of the instructions stored in the character generator. You have learned how to put data into a memory cell by using POKE. To find out what data is stored in a memory cell, you use the command PEEK. To demonstrate this command, type:

```
POKE 5ØØØ, 234
```

This stores '234' at address 5ØØØ, which is part of RAM. Now type:

```
X = PEEK(5ØØØ) : PRINT X
```

Note the brackets used after the word PEEK. POKE does *not* use brackets, but PEEK *does*. The screen displays the result of the PEEKing; the number '234' (see Fig. 7.2). POKE a different number to the same address, then use PEEK again to check that the number has been stored as directed. A quicker way of PEEKing is to type:

```
PRINT PEEK(5ØØØ)
```

Now to PEEK an address in the character generator. It has 4096 addresses, running from 32768 to 36863. It has so many addresses because it needs eight memory cells to store the data for each character. To find out what is in the first address, type:

```
PRINT PEEK(32768)
```

The result is '28'. If you try to POKE 32768 with any value, it has no effect, for the character generator is a kind of ROM. The number

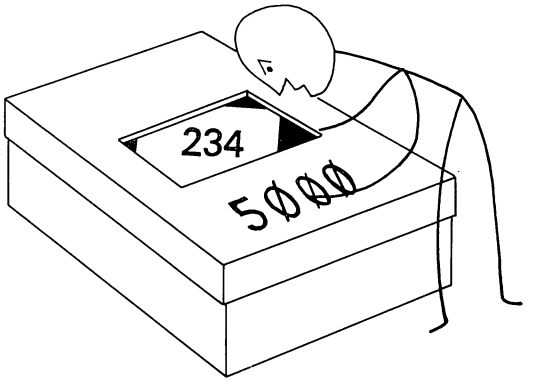


Fig. 7.2. PEEKing an address in RAM or ROM to find out what value is stored in it. Compare this with Fig. 3.1.

stored there remains unchanged at '28'. The first eight addresses of the generator hold the data for the first character. Use this program to find them:

```
1Ø FOR J = Ø TO 7
2Ø PRINT PEEK(32768 + J)
3Ø NEXT J
```

The numbers printed out, in order, are: 28 (as before), 34, 74, 86, 76, 32, 3Ø, and Ø. Figure 7.3 shows how they are interpreted. In the generator, these numbers are held as 8-digit (or 8-bit) binary numbers. We need to convert them to this form. The numbers are written down the left-hand side of the 8 × 8 grid. Across the top we have the 'powers of two', showing (in decimal) the values of the

Address	Value stored there (in decimal)	The powers of two: 128	64	32	16	8	4	2	1	The additions	
32768	28	Ø	Ø	Ø	1	1	1	Ø	Ø	16 + 8 + 4	= 28
32769	34	Ø	Ø	1	Ø	Ø	Ø	1	Ø	32 + 2	= 34
3277	74	Ø	1	Ø	Ø	1	Ø	1	Ø	64 + 8 + 2	= 74
32771	86	Ø	1	Ø	1	Ø	1	1	Ø	64 + 16 + 4 + 2	= 86
32772	76	Ø	1	Ø	Ø	1	1	Ø	Ø	64 + 8 + 4	= 76
32773	32	Ø	Ø	1	Ø	Ø	Ø	Ø	Ø	32	= 32
32774	3Ø	Ø	Ø	Ø	1	1	1	1	Ø	16 + 8 + 4 + 2	= 3Ø
32775	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	= Ø

Value stored there (in binary)

Fig. 7.3. Eight 8-digit (8-bit) binary numbers from successive addresses of the character generator.

binary digits in each column. Run along the row until you find a column heading which is *equal to or less than* the number on the left. In the first row, 128, 64 and 32 are all *more than* 28. Next is 16 which is *less than* 28. Write a '1' in that row in that column. Now subtract 16 from 28, leaving 12. The next column is headed '8', which is less than 12, so write a '1' in that column too. Subtract 8 from 12, leaving 4. The next column is headed '4', so write a '1' in that column too. Taking 4 from 4 leaves nothing, so the row is finished. In the second row, you need a '1' in the '32' column and the '2' column. In each row we are writing in '1's so that the headings of the columns with '1's in them add up to the number on the left. When you have finished write a '0' in all empty cells of the grid. The result is the set of eight binary numbers which represent a character. Which character? To find this out, take a second grid and shade the squares which correspond to the '1' squares of the first grid (Fig. 7.4). The result is a picture of the '@' symbol. This is the first character in the generator. You will note that in Table 6 (Appendix A), '@' has the screen code 0. Try PEEKing the memory cells for letter 'A' (from address 32776), using the same program and reconstruct 'A' on a grid.

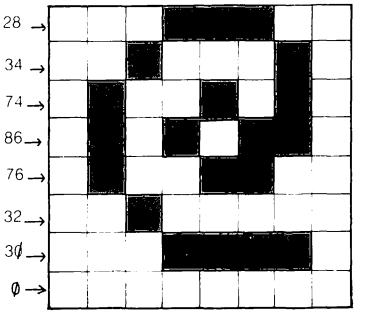


Fig. 7.4. If the '1's of Fig. 7.3 cause dark 'dots' to be displayed on the screen, the character '@' is produced.

From the above we can see that the instructions for displaying a number consists of eight 8-bit binary numbers, stored in eight successive addresses of the character generator. In each number, the digit '1' represents a dot printed on the screen. A '0' represents the absence of a dot. Having found this out, all we need to do to define a new character is to work out which binary numbers represent it. But wait! The character generator is like ROM; its memory is unchangeable; we cannot POKE a new set of values into any part of it. Obviously we are going to have to use RAM for storing the binary numbers. Then we shall have to arrange for the VIC to read its instructions from RAM instead of from the character generator.

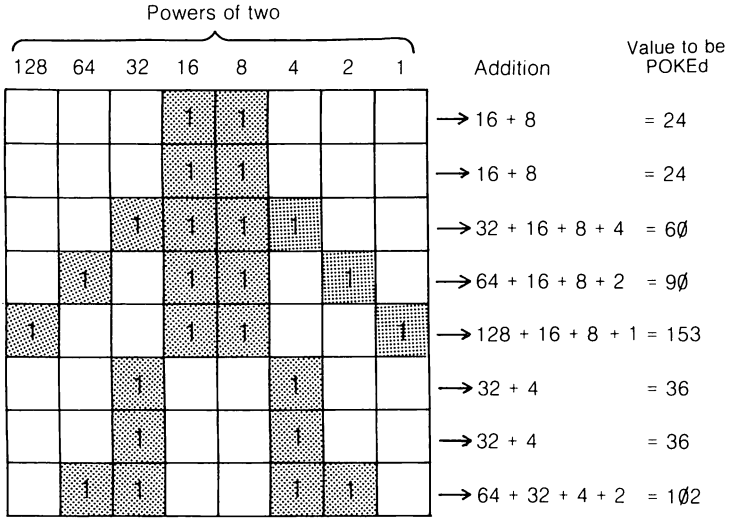


Fig. 7.5. Calculating the values needed for defining a new character. (*Note*: the program does this for you!).

Designing a new character

Figure 7.5 shows the first step. The character is sketched out on an 8 × 8 grid. Then, for each row, we add together the numbers at the heads of those columns which contain a shaded square. These values are to be POKEd into a suitable part of memory. It is best to use the higher addresses, so that our new 'character RAM' and the BASIC programs do not interfere with each other. Listing 7.2 shows a program which has a DATA line containing the numbers to be stored.

```
10 FOR J = 0 TO 7
20 READ X
30 POKE 7168+J,X
40 NEXT J
50 POKE 36869,255
60 PRINT"□"
70 END
100 DATA 24,24,60,90,153,36,36,102
```

Listing 7.2. Making the VIC display a new keyboard character.

Lines 1Ø to 4Ø READ the DATA line and POKE the numbers into eight successive memory cells, from 7168 onwards. Line 5Ø alters the value in one of the registers of the VIC chip, making the computer think that its character generator now starts at address 7168.

The screen clears when you RUN this program and some jumbled clumps of dots and dashes appear at the top right corner. This is the

computer printing its customary 'READY.' message. It is getting its character instructions from RAM now, and is using whatever jumble of '1's and '0's it happens to find there. To show that the program has been successful, press the '@' key. Each time you press it, the image of the little man is printed on the screen. The '@' key has been *redefined* to print a new character.

```
10 POKE 51,255:POKE 52,23:POKE 55,255:PO
KE 56,23
20 FOR J = 0 TO 1023
30 X = PEEK(32768 + J):IF J > 255 AND J
< 264 THEN X = 255
40 POKE 6144 + J,X
50 NEXT J
60 POKE 36869,254:POKE 36866,136:POKE 36
864,26:POKE 36867,144
70 PRINT"□"
80 INPUT X$
90 IF X$ = "E" THEN GOTO 220
100 POKE 36879,157
110 FOR J = 0 TO 7
120 FOR K = 0 TO 7
130 P = 38400 + 8*J+K
140 P = PEEK(P) AND 7:IF P = 6 THEN B(J,
N)= B(J,N) + 2↑(7-K)
150 NEXT K:NEXT J
160 PRINT"□":INPUT K$(N)
170 FOR J = 0 TO 7
180 POKE 5632 + 8*ASC(K$(N)) + J,B(J,N)
190 NEXT J
200 POKE 36879,27:B(8,N) = 5632 + 8*ASC(
K$(N)):N = N + 1
210 GOTO 70
220 POKE 36864,12:POKE 36866,150:POKE 36
867,174
230 FOR J = 0 TO 7:POKE 6400+J,0:NEXT J
240 PRINT"□"
250 PRINT K$(L);"  ";ASC(K$(L))
260 FOR J = 0 TO 8
270 PRINT B(J,L);" ";
280 NEXT J
290 GET Z$:IF Z$ = "" THEN GOTO 290
300 IF L = N - 1 THEN L = -1
310 L = L + 1: GOTO 240
```

Listing 7.3. Newkeys, a useful program to help you create your own new keyboard characters.

Defining new characters can be a tedious matter. Drawing out the grids and adding up the numbers takes a lot of time. The Newkeys program in Listing 7.3, allows you to design characters on the screen. It then tells you which numbers you have to POKE into RAM, and at which addresses to POKE them. Following the way this program works is a good way of starting to understand the Video Interface Chip. There is not space in this book to go into this in full detail, but the account below is a guide for those who want to experiment. If you would rather just use the program and not bother to understand it or the chip, go ahead to the section 'Using Newkeys' where full operating instructions are given.

Line 1ϕ alters data already stored at locations 51, 52, 55 and 56 so that the computer thinks the top of the RAM memory lies at 32767, instead of its usual position. This prevents it from trying to store a program or string variables in the same place as your character definitions. Lines 2ϕ to 5ϕ copy the contents of the first 1024 locations of the character generator ROM into RAM from address 6144 onward. In effect, we are setting up a new character generator in RAM. This is not a complete set of characters but only the capital letters, numerals, punctuation and mathematical symbols, and right side graphics characters. The 'space' (stored in locations 32324 to 32331) is *not* copied as a space but as a solid block. As you can see in line 3ϕ, these locations are filled with '255', giving 'all dots'. Why this is done is explained in a moment.

The computer must be printing in its usual blue colour before this program is RUN.

Line 6ϕ alters the contents of several of the registers in the Interface Chip. In order, the POKEs set: register 5 to change the base address for character memory from 32768 to 6144; register 3 to make the screen display eight columns wide instead of twenty-two; register 1 to widen the left-hand border so that the new eight-column screen is centred on the TV screen; register 4 to reduce the screen depth to eight rows instead of twenty-three. We now have a 'small screen', with eight rows and eight columns, ready for sketching the new character.

An INPUT statement at line 8ϕ makes the computer wait while we draw out the characters.

> The user presses SHIFT and CLR/HOME to put the cursor at the top left corner. Pressing the space-bar puts a block of solid blue on the screen; this is why we altered the 'space' to a block. Pressing the left-right CRSR keys moves on the cursor without printing blue. In this way the area is 'scanned', building up the design of the character in blue and white. When the design is complete, press RETURN.

The screen and border change colour. This is a visible indicator that something is happening, for nothing shows on the screen but the new design. In lines 11ϕ to 15ϕ, the computer scans the colour code RAM of the screen to find the colour of each block. The colour code memory begins at 38400 (see Fig. 3.3) but, since there are only 64 cells, it goes only as far as 38463. It PEEKs each memory cell (line 14ϕ) to find out the value of the lower three bits stored there. The way AND is used in this line differs from the ways explained in

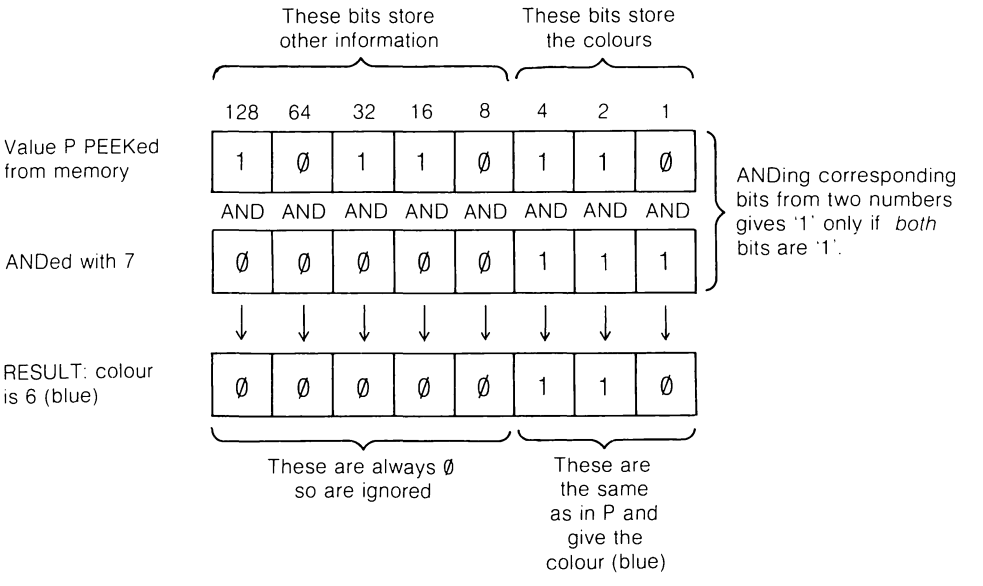

These bits store other information
These bits store the colours
128 64 32 16 8 4 2 1
Value P PEEKed from memory
1 Ø 1 1 Ø 1 1 Ø
AND AND AND AND AND AND AND AND
ANDed with 7
Ø Ø Ø Ø Ø 1 1 1
ANDing corresponding bits from two numbers gives '1' only if *both* bits are '1'.
RESULT: colour is 6 (blue)
Ø Ø Ø Ø Ø 1 1 Ø
These are always Ø so are ignored
These are the same as in P and give the colour (blue)

Fig. 7.6. Showing how line 14Ø of Newkeys finds out the colour of a block.

Chapter Six, and is explained in Fig. 7.6. As a result P takes the value of the colour code (Table 7 in Appendix A) for each block of the screen. If P = 6, the square is blue, and this means a 'dot' must be printed in the final character. The binary numbers we shall need are being added up in array B. This has J columns, corresponding to the 8 rows of the character and we add up a total in each, as each row of the screen is scanned. Array B has up to eleven columns allowing us to design up to eleven different characters in one session. B is not DIMensioned, so N cannot be greater than 1Ø.

The expression following the THEN of line 14Ø looks imposing, but is simply adding up the 'powers of 2' as each block of a row is scanned. For example for the left-hand block, K is zero, and, if this block is blue, B(N,J) is incremented by 2 to the power 7, or 128 (compare Fig. 7.3). For the next block, K is 1. If this block is blue, B(W,J) is incremented by 2 to the power 6, or 64.

At line 16Ø, the user is asked to INPUT a 'key'. This is the key to which the character is to be allocated. When this key has been pressed, (followed by RETURN), the micro POKEs the values from array B into the appropriate memory locations in the new character RAM. It finds where these are by using the ASCII code of the key just pressed(K$). The ASCII function appearing in line 2ØØ returns the ASCII code of any key. Since the memory cells allocated to the keyboard characters are in sets of eight, in order of ASCII code, the

correct addresses are easily calculated. These values for the newly designed character are POKEd into RAM, replacing the values for the keyboard character which were originally put there by lines 2ϕ to 5ϕ. For this purpose we usually choose keys such as '@', '$', or '/' which we do not need to use.

At line 21ϕ, the user is sent back to 7ϕ to design the next character. If no further characters are required, pressing "E" for exit sends the micro on to line 22ϕ to print out the details.

Line 22ϕ POKEs registers to: restore border width; make the screen 22 columns wide; and make it 23 rows deep. Line 23ϕ redefines the 'space' as a clear 'space' (all zeros).

Finally, the details of each character are printed out (see below).

Using Newkeys

(1) Set the printing colour to blue (CTRL with 6) if it is not already blue.
(2) RUN the program; it takes about 25 seconds to transfer the character codes to the new area of RAM. After this the 8 × 8 screen appears (white with cyan border).
(3) Press SHIFT with CLR/HOME; the cursor goes to top left of screen. It is speckled, but this does not matter.
(4) To build up the character you scan across the screen, row by row, from the top row down to the bottom row. To make a blue block, press the space-bar. To make a white block, press the left-right CRSR key.
(5) If you make a mistake, press SHIFT with CLR/HOME and start again from step (3). Do not use INST/DEL.
(6) At the end of the bottom row, the cursor moves off the bottom of the screen and is not seen. If you are satisfied with the design, press RETURN. If not, clear the screen and begin again from step (3).
(7) The border colour changes to green showing that the coding is being done. After this the cursor reappears, still looking speckled.
(8) Decide which key you want to be allocated to that character and press it (unshifted). You cannot use keys which INPUT will not accept; the colon and comma. It is best to avoid allocating the number keys, for this makes it hard to read the codes at step (10).
(9) Then press RETURN. The border becomes cyan and the screen white. You are back at step (3). Repeat the above for up to 10 more characters. If you have finished, press key E at this stage, and then RETURN.

(10) This takes you to the final printing out stage (the screen is full-sized now). For each character you see:

The new character.

The ASCII code of the key to be used for this character (this is in case you have forgotten the key – the program cannot, of course, print out the letter or symbol that used to belong to that key).

A list of eight numbers, which are those which have been poked into RAM. If you want to use the characters in another program, copy these numbers from the screen, in order.

The first address into which these have been POKEd. Copy this too, if required.

Press any key, to view each character and its details in turn. When you have finished designing characters, you will have a keyboard in which some of the keys give their usual characters, while others give the characters you have designed. You can use this for typing or for designing graphic layouts on the screen. The characters can be used in displays for games programs and others. If you want to return to the normal use of the keyboard, type this line:

```
POKE 36869,24Ø
```

When you press RETURN, any of your new characters which are already on the screen instantly revert to the key characters to which they have been assigned. The keyboard behaves normally, making it much easier to type programs. However, the new characters are still there (unless you switch off or press RUN/STOP with RESTORE) and you can get back to them at any time by typing:

```
POKE 36869,254
```

If you want to write a program to use these characters, delete the Newkeys program. You will find that you now have 2K of RAM available for your program. At any stage of programming, or even during the running of the program, you can flip from the normal set of characters to your new set or back again by using one of the two POKE commands just given. While writing the program in normal mode, use your specially-defined keys where you want the new characters to appear. While you are programming, they display with their normal 'key-top' form, but when you have POKEd to return to the newly-defined character set, your own designs will reappear.

Later in this chapter there is more about using these characters in programs. By the way, do not use Text Mode or reversed characters, for these are not present in your RAM character set.

Multikeys

The normal VIC characters and the ones defined by the Newkeys program are in two colours:

(1) The *foreground colour* (sometimes referred to as the 'printing' colour), which is normally blue.
(2) The *background colour* (sometimes referred to as the 'screen' colour), which is normally white.

By using a special VIC graphics mode called *Multicolour Mode*, we can add two further colours to a graphics character:

(3) The *border colour*; the same colour as the border of the TV screen.
(4) An *auxiliary colour*; any fourth colour.

The foreground colour and the border colour can be any of the eight colours listed across the top of Table 5 in Appendix A. The background colour and auxiliary colour can be any of the 16 colours listed down the left of Table 5.

As might be expected, a gain of colours means a loss in some other direction. In each row of the character the dots are in four pairs and members of each pair must both be the same colour. In effect, each row consists of four 'double' dots. The definition obtained from multicolour graphics is not as sharp as that of ordinary two-colour graphics, at least when viewed on an ordinary TV screen. Unless you have a colour monitor, it is best to confine your designs to bold ones.

The data for a multicolour character is stored as eight 8-bit binary numbers, as were the dots for the two-colour characters. The difference is that when we had only two colours we could represent these by '0' or '1', but now we have four colours we need two bits to represent them. The four colour codes are,

00 background colour
01 border colour
10 foreground colour
11 auxiliary colour

The bits of each binary number are taken two at a time and interpreted as above. Figure 7.7 shows how it is done. For example, to get a strip of foreground colour down the left side of the character the first two bits of each code number must be '10'.

Defining a character thus means deciding which areas of it shall take the four available colours. We then have to POKE the Video

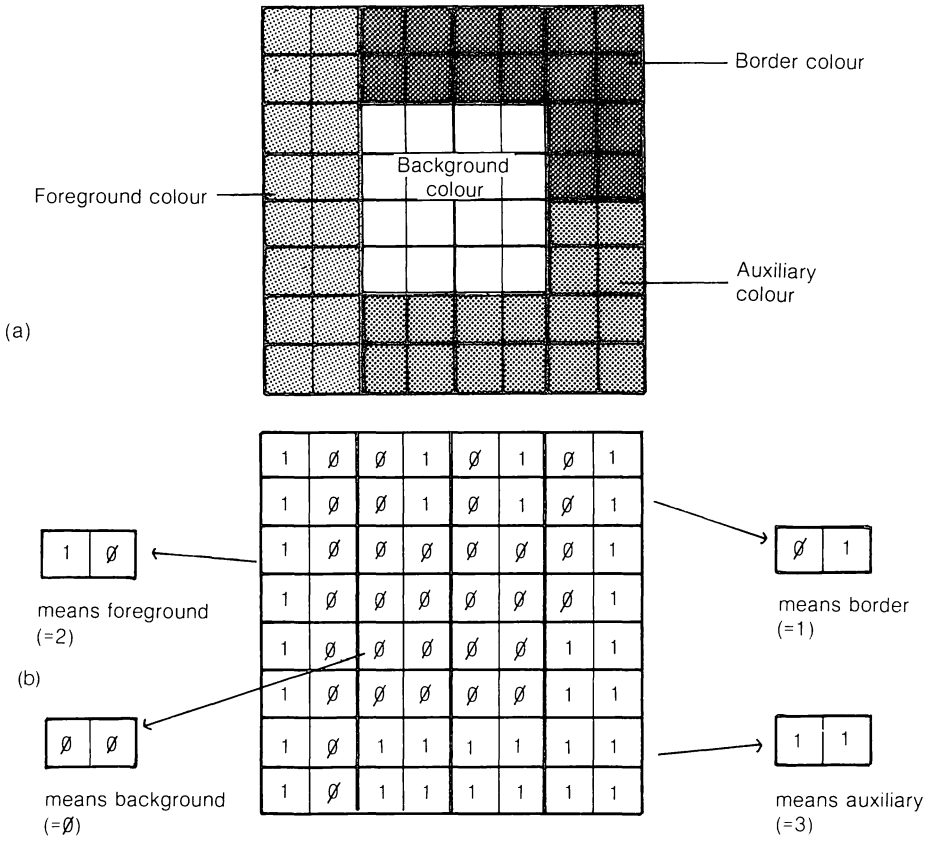

Border colour
Background colour
Foreground colour
Auxiliary colour
(a)
1 Ø Ø 1 Ø 1 Ø 1
1 Ø Ø 1 Ø 1 Ø 1
1 Ø Ø Ø Ø Ø Ø 1
1 Ø Ø Ø Ø Ø Ø 1
1 Ø Ø Ø Ø Ø 1 1
1 Ø Ø Ø Ø Ø 1 1
1 Ø 1 1 1 1 1 1
1 Ø 1 1 1 1 1 1
1 Ø
means foreground (=2)
Ø 1
means border (=1)
(b)
Ø Ø
means background (=Ø)
1 1
means auxiliary (=3)

Fig. 7.7. A design for a multicolour character. (a) Layout of the coloured areas. (b) Corresponding bit pattern; in each row the bits are filled in as pairs.

Interface Chip to set the border and screen (background) colours, as explained in Chapter Three. We set the foreground colour by using CTRL with a number key. We set the auxiliary colour by POKEing its code to another register in the VIC. Incidentally, all multicolour characters displayed on the screen at the same time must necessarily employ the same four colours. Multicolour characters may be used mixed with ordinary characters; either those of the regular set or those you have defined by means of Newkeys.

The Multikeys program (Listing 7.4) takes the tedium out of defining multicolour characters. It works in much the same way was Newkeys (Listing 7.3). If you want to *use* it rather than understand it, move on to the section 'Using Multikeys', for full instructions. The description below does not repeat details already covered by the description of Newkeys. You will see that this program does not have all its lines numbered at 'even tens'. This is because so many of

the lines are the same as for Newkeys. If you have typed in Newkeys, it does not take long to revise a few lines and add some more to convert it to Multikeys.

```
10 POKE 51,255:POKE 52,23:POKE 55,255:PO
KE 56,23
20 FOR J = 0 TO 1023
30 X = PEEK(32768 + J):IF J > 255 AND J
< 264 THEN X = 255
40 POKE 6144 + J,X
50 NEXT J
51 PRINT"□"
52 INPUT"BORDER";BO
53 INPUT"BACKGROUND";BA
54 BO = BO - 1:BA = BA-1:IF BA > 8 THEN
BA = BA - 2
56 INPUT"FOREGROUND";FO
57 INPUT"AUXILIARY";AU
58 FO = FO -1:AU = AU-1:IF AU > 8 THEN A
U = AU - 2
60 POKE 36869,254:POKE 36866,136:POKE 36
864,26:POKE 36867,144
70 PRINT"□"
80 INPUT X$
90 IF X$ = "E" THEN GOTO 220
100 POKE 36879,157
110 FOR J = 0 TO 7
120 FOR K = 0 TO 7 STEP 2
130 P = 38400 + 8*J+K
140 P = PEEK(P) AND 7
142 IF P = BO THEN B(J,N) = B(J,N) + 4↑(
3-K/2)
144 IF P = FO THEN B(J,N) = B(J,N) + 2*4
↑(3-K/2)
146 IF P = AU OR P = AU - 8 THEN B(J,N)
= B(J,N) + 3*4↑(3-K/2)
150 NEXT K:NEXT J
160 PRINT"□":INPUT K$(N)
170 FOR J = 0 TO 7
180 POKE 5632 + 8*ASC(K$(N)) + J,B(J,N)
190 NEXT J
200 POKE 36879,27:B(8,N) = 5632 + 8*ASC(
K$(N)):N = N + 1
210 GOTO 70
220 POKE 36864,12:POKE 36866,150:POKE 36
867,174:POKE 36878,16*AU
221 IF FO = 0 THEN PRINT"■"
222 IF FO = 1 THEN PRINT"■"
223 IF FO = 2 THEN PRINT"■"
224 IF FO = 3 THEN PRINT"■"
225 IF FO = 4 THEN PRINT"■"
226 IF FO = 5 THEN PRINT"■"
227 IF FO = 6 THEN PRINT"■"
228 IF FO = 7 THEN PRINT"■"
229 POKE 36879,8 + 16*BA  + BO
230 FOR J = 0 TO 7:POKE 6400+J,0:NEXT J
240 PRINT"□"
250 PRINT K$(L);"  ";ASC(K$(L))
255 POKE 38422,FO+8
260 FOR J = 0 TO 8
265 POKE 38422,FO+8
270 PRINT B(J,L);" ";
280 NEXT J
290 GET Z$:  IF Z$ ="" THEN GOTO 290
300 IF L = N - 1 THEN L = -1
310 L = L + 1:GOTO 240
```

Listing 7.4. Multikeys, a useful program to help you design multicolour (4-colour) graphics characters.

Lines 51 to 57 ask the user to key in the colours which are required. The queries are answered by typing one of the number keys 1 to 8 (see instructions in the next section). The variables BO, BA, FO and AU are used for the colour codes.

At line 8Ø the computer waits for the user to sketch in the design. This time, the space-bar is used to enter blocks of one of the four colours. The CRSR key is not used. The colour of the blocks is controlled by pressing CTRL and a number key 1 to 8 in the normal way.

Line 12Ø steps in twos across each row of the screen to detect the colours. Both members of a pair of blocks are supposed to be of the same colour and, if they are not, this fact is ignored. The colour already on a block is detected in line 14Ø and one of lines 142 to 146 adds the appropriate amount to the value being built up in B(J,N).

If a block is of the background colour (code ØØ), nothing is added. If the block is in border colour (code Ø1) the amount added is a power of 4, depending on the position of the block (see Fig. 7.8). If the block is of foreground colour (code 1Ø, equals 2 in decimal), the amount added is twice the power of 4. If the block is the auxiliary colour (code 11, equals 3 in decimal), the amount added is three times the power of 4.

Powers of four

64	16	4	1	Addition	Value to be POKEd
1 Ø = 2	Ø 1 = 1	Ø 1 = 1	Ø 1 = 1	→ 2×64+16+4+1	=149
1 Ø = 2	Ø 1 = 1	Ø 1 = 1	Ø 1 = 1	→ 2×64+16+4+1	=149
1 Ø = 2	Ø Ø = Ø	Ø Ø = Ø	Ø 1 = 1	→ 2×64+Ø+Ø+1	=129
1 Ø = 2	Ø Ø = Ø	Ø Ø = Ø	Ø 1 = 1	→ 2×64+Ø+Ø+1	=129
1 Ø = 2	Ø Ø = Ø	Ø Ø = Ø	1 1 = 3	→ 2×64+Ø+Ø+3×1	=131
1 Ø = 2	Ø Ø = Ø	Ø Ø = Ø	1 1 = 3	→ 2×64+Ø+Ø+3×1	=131
1 Ø = 2	1 1 = 3	1 1 = 3	1 1 = 3	→ 2×64+3×16+3×4+3×1	=191
1 Ø = 2	1 1 = 3	1 1 = 3	1 1 = 3	→ 2×64+3×16+3×4+3×1	=191

Fig. 7.8. Working out the values for the multicolour character of Fig. 7.7. The Multikeys program does all this for you.

Several characters can be defined, as in Newkeys, until the user presses "E". Then, at line 22Ø, the last item in the line puts the

auxiliary colour into register 15 of the Interface Chip. This register is the one we have already used for controlling the volume of the sound generators. The lower 4 bits of this register (Fig. 7.9) control volume, but the upper 4 bits hold the auxiliary colour. To put the value of AU into these upper 4 bits, we multiply it by 16, as in line 22Ø.

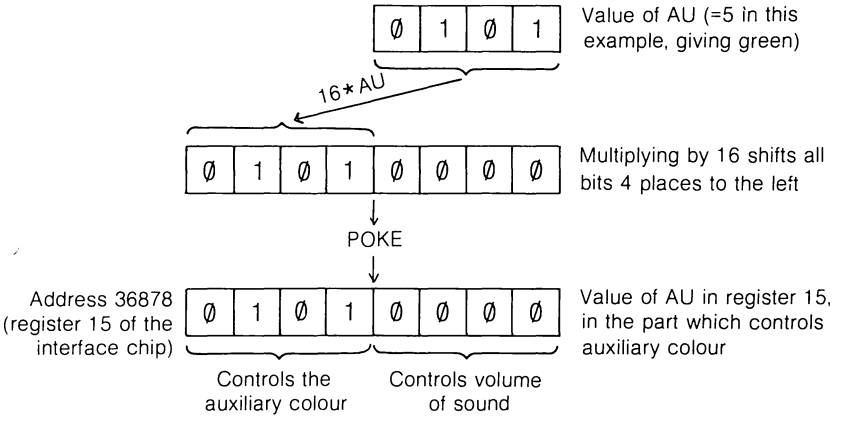

Ø 1 Ø 1
Value of AU (=5 in this example, giving green)
16*AU
Ø 1 Ø 1 Ø Ø Ø Ø
Multiplying by 16 shifts all bits 4 places to the left
POKE
Address 36878 (register 15 of the interface chip)
Ø 1 Ø 1 Ø Ø Ø Ø
Value of AU in register 15, in the part which controls auxiliary colour
Controls the auxiliary colour
Controls volume of sound

Fig. 7.9. Controlling the auxiliary colour (line 22Ø of Multikeys). AU can have any value from 0 (0000) to 15 (1111).

Lines 221 to 229 set the chosen foreground colour. Line 229 sets the chosen background and border colours. The display routine is

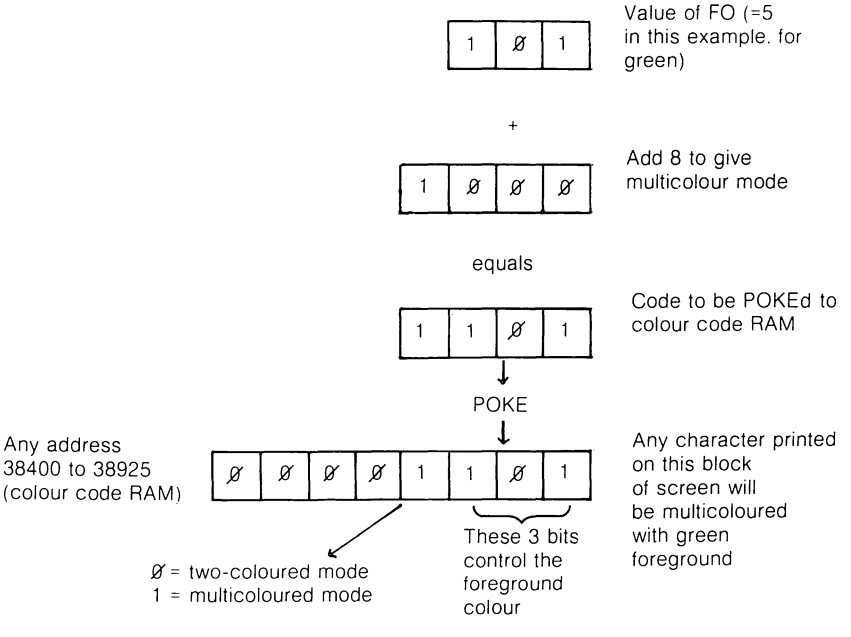

1 Ø 1
Value of FO (=5 in this example, for green)
+
1 Ø Ø Ø
Add 8 to give multicolour mode
equals
1 1 Ø 1
Code to be POKEd to colour code RAM
POKE
Any address 38400 to 38925 (colour code RAM)
Ø Ø Ø Ø 1 1 Ø 1
Any character printed on this block of screen will be multicoloured with green foreground
These 3 bits control the foreground colour
Ø = two-coloured mode
1 = multicoloured mode

Fig. 7.10. Controlling the foreground colour in multicolour mode (line 265 of Multikeys). FO can have any value from 0 (000) to 7 (111).

almost the same as with Newkeys. The only difference in programming is that to print a multicolour character we have to POKE to add 8 to contents of the corresponding address in colour code memory (Fig. 3.3). The character is printed on the second row in the first column, which has the address 38422. We POKE this address with the foreground colour plus 8. Figure 7.10 shows what happens. If you do not set the colour code memory address in this way, the character is displayed in two-colour mode, and this generally looks somewhat odd!

Using Multikeys

(1) It does not matter what the printing (foreground) colour is to begin with.
(2) RUN the program. After 25 seconds you are asked to type in:

Border colour: key 1 to 8, referring to the colours marked on the fronts of the keys.

Background colour: key 1 to 8 or key 1 (to indicate that you want a light colour), followed by the number of the light colour you want. 'Light black' (11) gives orange. 'Light white' (12) gives light orange. 'Light red' (13) gives pink, etc.

Foreground colour: key 1 to 8.

Auxiliary colour: key 1 to 8 or 11 to 18, as for background colour.

Write these colours down so that you will remember them later.

Note that with this program it is not possible to have both a colour and its lighter version in the same character. Now the 8 × 8 screen appears.
(3) Press SHIFT with CLR/HOME; the speckled cursor goes to the top left of the screen.
(4) To build up the character, you scan the screen, row by row, from the top row down to the bottom row. Use the space-bar for doing this (*not* the CRSR key). Press CTRL with a key 1 to 8, to set the colour for printing each block. Remember that both blocks of a pair *must* be the same colour. If you make a mistake, press SHIFT with CLR/HOME and begin again from step (3).
(5) At the end of the bottom row, the cursor moves off the bottom of the screen. If you are satisfied with the design, press RETURN. If not, clear the screen and start again, from step (3).

(6) The border colour changes to green, showing that coding is being done. After this, the cursor reappears.
(7) Decide which key is to be allocated to the character and press it (do not use colon or comma). Then press RETURN.
(8) The border then becomes cyan and you are back at step (3). Repeat the above for up to 10 characters, if you wish. When you have finished, press "E" and then RETURN.
(9) This takes you to the display stage. For each character you see:

The new character.
The ASCII code.
The 8 codes.
The first address.

The way to interpret these is explained on page 93. Press any key to view the next character.

Using the characters

Having defined characters and allocated them to the keyboard, you can use the keyboard for typing the characters on to the screen, or you can incorporate the new characters into PRINT statements. If you do not want to define any more characters, delete the program. This leaves you with about 2K of memory for program storage.

At this stage, you can write programs based on the characters you have just designed. In Chapter Eleven there are many examples of programs which do just this.

When you switch off, or if you press RUN/STOP with RESTORE, the newly acquired characters are lost. Provided that you copied down their codes at the end of the Newkeys or Multikeys programs (Listings 7.3 and 7.4), they are not lost for good, for you still have the codes. You can incorporate these into any program which you write later, so as to redefine as many of the keys as you want as special characters. Characters defined on different occasions can be brought together into one program. Some may have been created by Newkeys, and others by Multikeys. You can soon build up a 'library' of you favourite character definitions.

If your interest lies in games programs, your collection may consist mainly of designs for aeroplanes, spacecraft, monsters and the like or, perhaps, designs for pieces used in chess and other more serious games. If you are wanting to use the VIC for ecucational programming, you will find it easy to design the '$_2$', the '$_4$' and the

other 'inferior-position' characters used in chemical formulae, such as in H_2SO_4. For mathematics, you can design 'superior-position' characters, such as '^{2}' and '^{3}'. There is also plenty of scope for other mathematical symbols, such as Σ, °, and $\approx$.

In most programs you will want to be able to use both the newly defined characters together with the alphabet, numerals, symbols, and many of the ordinary graphics characters of the VIC's keyboard. The way to incorporate your new characters along with most of the normal characters is described in Chapter Eleven.

To summarise

In this chapter, you have found out how to:

- use the PRINT command to produce graphics displays
- use the TAB command to place PRINTed strings where we want them on the screen
- use the ASCII function
- use the CHR$ function
- redefine keys to print two-colour characters
- redefine keys to print multicolour characters.

Chapter Eight

Sorting It Out

Before we get down to the sorting, we will look at a few more operations which can be performed with strings. We have already seen that strings can be added together. This is not addition in the arithmetical sense, for all that is done is to take two or more strings and join them end to end. Can we subtract strings too? Not really, but there are ways of removing parts of strings, which might possibly be considered to be the equivalent of subtraction. Try this:

```
1Ø INPUT A$
2Ø INPUT N
3Ø B$ = LEFT$(A$,N)
4Ø PRINT B$
```

At the first INPUT, you type in a word. For example, you could type "ANYTHING". At the second INPUT you type a number; preferably one which is less than the number of letters in the word. If you type '3', for example, line 3Ø makes B$ equal to the first 3 letters of A$. The screen displays "ANY". LEFT$ is a *string function.* It requires two *arguments* (as we call the items placed in brackets after the word LEFT$). The first argument is the name of the string which is to be worked on. The second argument is the number of letters to be taken from that string, counting from the left. RUN the program a few more times, until you are used to the idea of LEFT$.

In the program above. B$ is simply an intermediate stage, so lines 3Ø and 4Ø could be replaced by:

```
3Ø PRINT LEFT$(A$,N)
```

RIGHT$ has a similar function, except that it operates from the right:

```
1Ø INPUT A$
2Ø INPUT N
3Ø PRINT RIGHT$(A$,N)
```

If you enter "ANYTHING" and then '5', you will see "THING" displayed on the screen.

The third member of this trio of string functions is MID$. As you might expect from its name, this one takes a slice (or sub-string) out of the middle of a string. This needs an extra value in the arguments. The first argument is the string, as before. The second is the position in the string from which the sub-string is to begin. The third is the number of letters which the sub-string is to contain. For example, if A$ is "ANYTHING", then MID$(A$,4,4) gives "THIN".

The Clock program in Listing 8.1 illustrates these string functions in action.

```
10 INPUT"HOURS";H$
20 INPUT"MINUTES";M$
30 TI$ = H$ + M$ + "00"
40 IF RIGHT$(TI$,2) <> "00" THEN GOTO 40
50 PRINT"□"
60 H$ = LEFT$(TI$,2)
70 M$ = MID$(TI$,3,2)
80 PRINT"THE TIME IS ":PRINT
90 PRINTTAB(8)H$;":";M$
100 FOR J = 1 TO 1000: NEXT J
110 GOTO 40
```

Listing 8.1. The Clock program illustrates ways of handling strings.

The hours and minutes are entered as strings, H$ and M$. These must be keyed in as two digits. For example, if the time is five minutes past two, the hours are '02' and the minutes are '05'. You could improve the program by writing routines which would allow single digits to be entered.

Line 30 builds up a six-digit string as required by TI$. Seconds are not entered, so the final two digits of the string are '00'. Line 40 examines TI$ repeatedly to see if its last two digits are '00'. This happens whenever the time is an *exact* number of minutes from the starting time. RIGHT$ shows its usefulness here.

The program waits at line 40 until a minute has just begun, then the micro goes to line 50 through to line 90 to display the time. The hours and minutes are taken from TI$ by using LEFT$ and MID$. We need the third and fourth characters of TI$ for the hours, so need to take two characters, beginning with the third. Line 100 inserts a short delay into the program, for if the micro goes back to line 40 *immediately*, the last two digits of TI$ may still be '00' and a flickering display is produced.

This program illustrates several other points about handling string variables. As it stands, it displays the time according to its 24-hour clock. Could this be changed to a 12-hour clock? What we need to do is to test H$ to see if it holds a value greater than 12. If so, we

have to subtract 12 from it. String variables cannot be treated as if they are numeric variables, even if, as in this program, they hold numbers. There is a function VAL which converts a string to a numeric variable. There is also a function STR$ which performs the opposite operation. This additional line for the Clock program uses both:

```
65 IF VAL(H$) > 12 THEN H$ = STR$(VAL(H$) - 12)
```

The VAL(H$) following the IF converts H$ into a number. If H$ is "15" then VAL(H$) is 15. It can now be compared with the number 12. If it is greater than 12, then H$ is to be changed. This is how it is done. The STR$ function has the argument (VAL(H$) − 12). If the value of H$ is 15, it is reduced by 12, giving 3. Then STR$ turns this number 3 into a string "3". H$ is made equal to this new string. As a result the program displays the time as 3.23 instead of 15.23.

VAL and STR$ are a useful pair of functions for handling strings which contain numbers. If VAL is used with a string which has *mixed* numbers and letters, it reads through the string from left to right, stopping when it comes to a character which is not a number. Thus:

If A$ = "23 High Street", VAL(A$) is 23 (useful for picking street numbers out of addresses).

If A$ = "ANYTHING", VAL(A$) is Ø, for there are no numbers before the first non-number character.

If A$ = "1 APRIL 1983", VAL(A$) is 1, and the "1983" beyond the name of the month is ignored.

Three non-number characters which *are* accepted by VAL are '+', '−' (minus) and '.' (decimal point).

If you have added line 65 to the Clock program, you may have noticed that it sometimes displays the hour with a leading zero (e.g. Ø2.35) and sometimes without (e.g. 2.35). Before noon, VAL(H$) is less than 12, so line 65 does not operate. H$ is a two-character string, with a leading zero because it has been obtained by taking two characters out of TI$. After noon, line 65 operates and (VAL(H$) −12) produces a *number* less than 12. This has no leading zero, for it is a number, not a string. Then STR$ converts it to a string as it *stands*, without the leading zero.

If you prefer not to have a leading zero, it can easily be removed from H$ by using RIGHT$:

```
63 IF VAL(H$) < 1Ø THEN H$ = RIGHT$(H$,1)
```

Before leaving the clock program, you might like to improve it in various other ways. It is easy to incorporate an alarm facility. The user is asked to type in the time when the alarm is required. This could be stored as A$, a six-digit number, ready to match against TI$. When TI$ = A$ the program jumps to a subroutine to flash the colours of the screen and border, or to bring the sound generators into action. Another modification is to make it print A.M. or P.M. according to whether it is before or after noon. A more ambitious addition to the program would be to get it to chime the hours – and perhaps the quarters too!

Longspell

This is a spelling game which demonstrates another string function, LEN. This is used to find out how many characters are present in a string. If A$ = "ANYTHING", for example, LEN(A$) = 8. The number of characters includes numbers, punctuation marks and spaces so that, for example, if A$=" HALF-PAST 2 ", with a space at the beginning and at the end, LEN(A$) = 13.

```
10 FOR J = 1 TO 6
20 N = INT(RND(1)*90)
30 IF N < 65 THEN GOTO 20
40 A$(J) = CHR$(N)
50 NEXT J
60 PRINT"□":FF = 0
70 FOR J = 1 TO 6
80 PRINTA$(J);"   ";
90 NEXT J
100 PRINT:PRINT"YOUR WORD?"
110 PRINT:INPUTW$
120 FOR J = 1 TO LEN(W$)
130 F = 0
140 T$ = MID$(W$,J,1)
150 FOR K = 1 TO 6
160 IF A$(K) = T$ THEN F = 1
170 NEXT K
180 IF F = 0 THEN FF = 1
190 NEXT J
200 IF FF = 1 THEN PRINT:PRINT"OTHER LET
TERS!":GOTO 230
210 PRINT:PRINT LEN(W$);"LETTERS"
220 IF LEN(W$) > M THEN M = LEN(W$):PRIN
T:PRINT"THE LONGEST WORD YET."
230 GET X$: IF X$ = "" THEN GOTO 230
240 IF X$ = "N" THEN GOTO 10
250 GOTO 60
```

Listing 8.2. Longspell, a simple but mind-stretching word game.

'Longspell' (see Listing 8.2) begins with a routine to choose six letters of the alphabet, at random. Line 2Ø selects a random number between 1 and 89, while line 3Ø rejects any numbers below 65. This leaves us with numbers between 65 and 89. If you look at Table 10 in

Appendix A, you will see that numbers in this range are the ASCII codes for letters of the alphabet from A to Y. Using CHR$, the random number causes a random letter to be put into array A$, which eventually holds six such letters.

It is just as well to leave out Z, since the aim of the game is to make up the longest word that you can, using the six letters randomly chosen. The same letter can be used as many times as you like in building up the word. The game may be played by two or more persons, to see who can make up the longest word from the given letters. If you play alone, you can try to beat your own record for the longest word. A variation is to try to make up the longest sentence from the given letters.

Type in the word at line 11Ø. Now the computer checks through your word (W$), letter by letter, to see that you are not cheating by using letters which have not been chosen. The checking is done in a FOR...NEXT loop. LEN(W$) in line 12Ø is the length of your word, so the computer knows how many letters there are to check. Each time round the loop, it picks out a letter from your word, using MID$ (line 14Ø) and calling this T$. It has to check T$ against each letter in array A$; it *must* match T$ to one OR another of these. It has to do this for every letter of your word in turn and make certain that the first letter of your word AND the second, AND the third, AND so on, is acceptable. The whole operation could have been written out as logical statements, with plenty of ORs and ANDs, but there would be so many that they would not fit on one line. Two flags, F and FF, are used to take care of the logic. The way they work is explained in Fig. 8.1.

Line 2ØØ rejects your word if it contains a letter which is *not* in array A$. At line 21Ø, LEN$ is used again to find out how many letters your word contains. Line 22Ø shows how we keep a running score of the maximum value of a series of numbers. The maximum number is called M, and this is zero when the program begins. But whenever any value of LEN(A$) exceeds M, M takes this value. In this way M gradually increases when you type in longer words. Line 23Ø waits for you to read what is displayed. Press any key except 'N' to have another try at finding more words using the same selection of letters. Type 'N' to make the computer choose a new set.

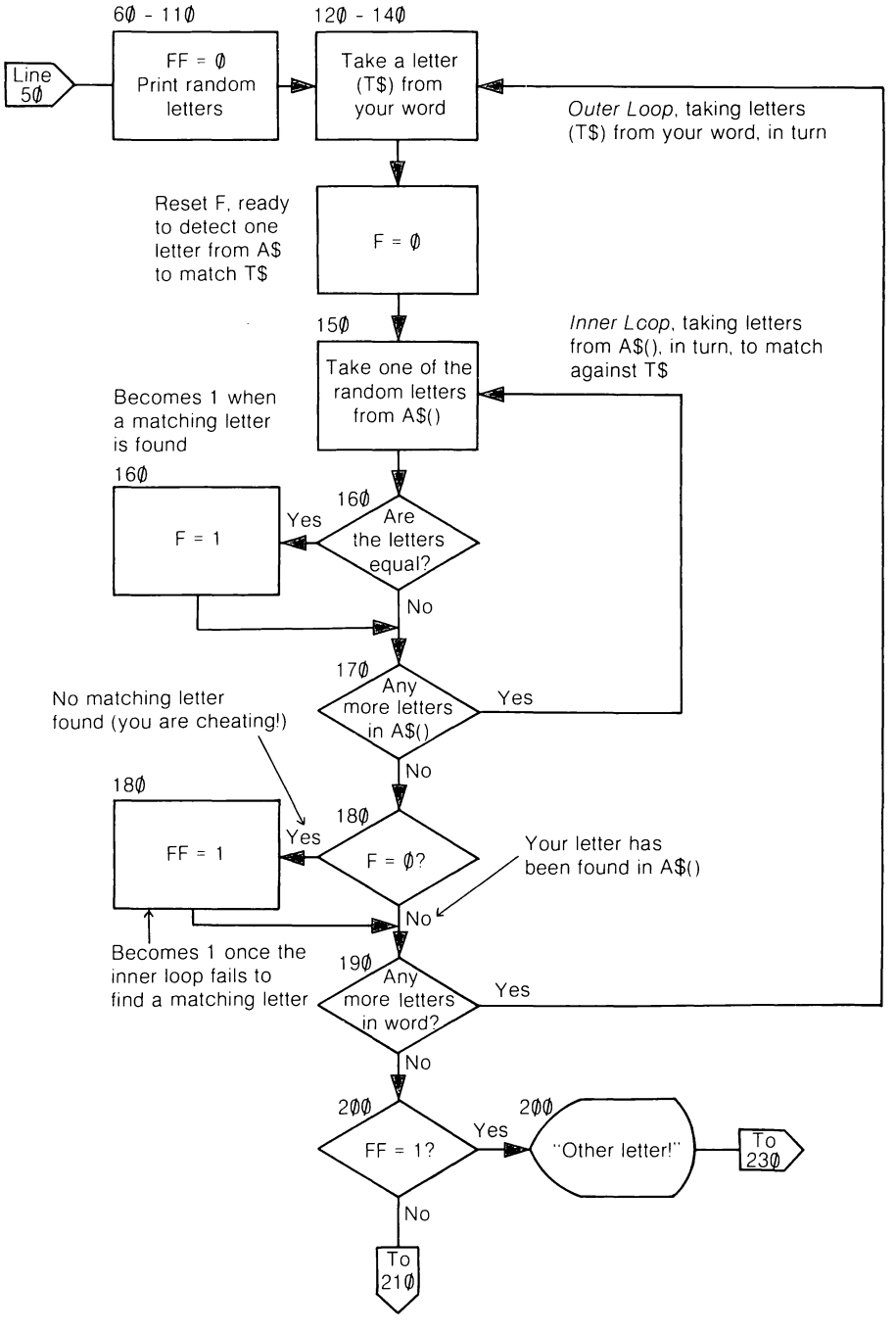


Fig. 8.1. Checking your word in Longspell.

Is love greater than hate?

At their present state of development, computers are not able to deal with philosophical questions such as that posed in the heading of this section. We could, of course, use LEN to ask if the word 'love' is longer than the word 'hate', but this is hardly worth doing, since the answer is obvious already. Let's see how the micro answers this question by using this program:

```
1Ø INPUT A$
2Ø INPUT B$
3Ø IF A$ < B$ THEN PRINT A$:PRINT B$:STOP
4Ø PRINT B$:PRINT A$
```

RUN the program. At the first INPUT, type "LOVE" and at the second, type "HATE". As you can see from the listing above, the program makes the computer print the lesser string first, followed by the greater string. The answer is clear – the computer displays:

```
HATE
LOVE
```

Clearly love is greater than hate. If you type "HATE" first and "LOVE" second, the result is the same. Now try typing in other pairs of words, such as "THIS" and "THAT", "ANYTHING" and "NOTHING", "PROGRAMMER" and "PROGRAMMABLE", and several other pairs. Can you see how the computer decides which is the greater?

The program is a very simple one for sorting two words into alphabetical order. The computer compares the two words and takes the one which comes second in alphabetical order as the greater. Try the program with a pair such as "ANT" and "ANTARCTIC". These words are sorted with ANT first, and ANTARCTIC second with is just the way we arrange such pairs of words in an alphabetical list. Obviously, this sorting ability is likely to be a useful one for handling lists of people's names, and catalogues and indexes of all kinds.

The program in Listing 8.3 keeps a list of words, in an array W$. When any new word is added to the list, the computer sorts through the list and inserts it in its correct position. This is really a demonstration program, but the main routines of it can easily be adapted and extended to make it of practical use. It illustrates several programming techniques.

The program begins by setting up an array W$, with space for 21

```
10 DIM W$(21)
20 W$(1) = "ZZZ"
30 PRINT"■"
40 PRINT:INPUT"WORD";X$
50 PRINT"■":J = 1
60 IF X$ > W$(J) THEN J = J + 1:GOTO 60
70 FOR K = 21 TO J + 1 STEP - 1
80 W$(K) = W$(K-1)
90 NEXT K
100 W$(J) = X$
110 N = N + 1
120 X = FRE(0)
130 FOR L = 1 TO N
140 PRINT W$(L)
150 NEXT L
160 IF N < 21 THEN PRINT:GOTO 40
170 PRINT:PRINT"■LIST FULL";
180 GOTO 180
```

Listing 8.3. A program which sorts names in alphabetical order.

words (ignoring W$(Ø) – we shall see why in a moment). Then a 'final word' ("ZZZ") is put into the array at the first position (line 2Ø). After you have typed in a word (X$), the computer examines each word in the list in turn. The variable J is used to indicate the word being looked at. In line 6Ø the computer compares X$ with W$(J), to see if X$ comes after W$(J) in alphabetical order. If it does, J is incremented and the comparison is made again for the next word in the array. Eventually it is found that X$ is not greater than the word in W$(J). This means either that X$ is the same as W$(J) or comes before W$(J).

The next step is to insert X$ in the list, in position J. This is done by moving all the remaining words (from J + 1 to the end of the list) one position further down in the list. Figure 8.2 shows what happens. The program begins at the *bottom* of the list, transferring into each position, (W$(K)), the word stored in the position above, (W$(K − 1)). Then X$ is stored at W$(J).

Line 11Ø keeps count of the number of words in the list and, in lines 13Ø to 15Ø, the words are printed out in order, so that you can see how sorting has progressed. Line 12Ø includes a BASIC word that has not been mentioned before. The most frequent use for this word is to find out how much memory you have left for your program. If you type:

```
PRINT FRE(Ø)
```

and press RETURN the computer displays the number of bytes of memory which are still free for programs. The zero after FRE is a dummy; it just has to be there but serves no purpose. Issuing the FRE command has the effect of making the VIC look through memory to see how much space there is. While doing this, it may find

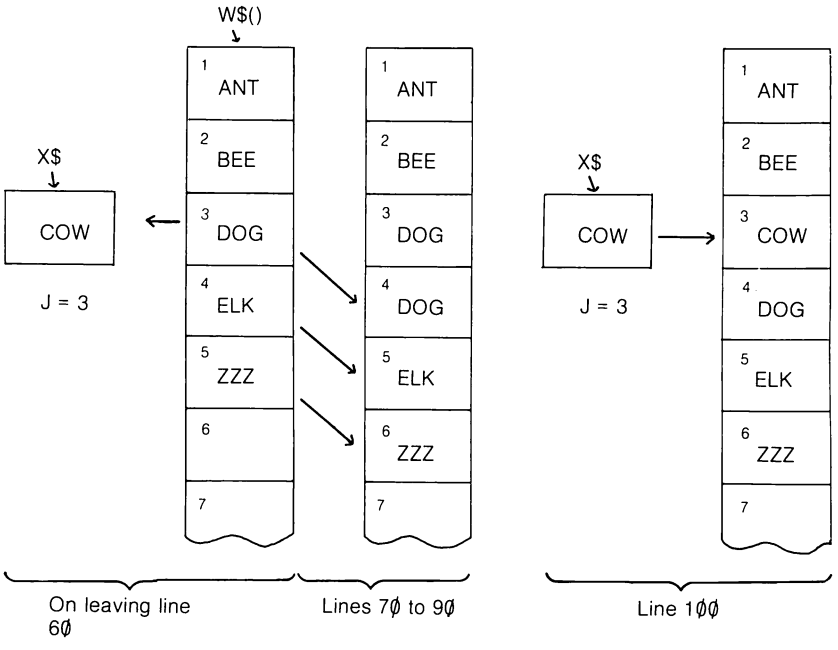


Fig. 8.2 Inserting a new item **X$** in its correct position in **W$()**.

lots of space taken up by strings which are no longer in use. It goes through a routine to tidy up the memory space in which strings are stored, getting rid of the unused strings and so freeing more memory for programs. The operation in line 12Ø forces the computer to tidy up its memory. This is necessary when lots of strings are being moved around and being given new names, for the memory used to store the old strings (before they were renamed) is still being occupied. FRE clears up the 'bits and pieces', allowing many strings to be moved and stored in memory without causing an "OUT OF MEMORY" error message. The FRE function is not really necessary in this program, for there is plenty of room in which to move and store twenty words, but it is included to remind you that it may be important in a sorting program which has to handle many words.

Sorting numbers

If the program above is modified slightly, it can be used for sorting numbers instead of words. The changes required are:

Delete the '$' from all lines
alter line 2ϕ to: W(1) = 1E38 (Almost the largest number the computer can hold)
alter line 4ϕ to PRINT:INPUT"NUMBER";X

The program sorts by *inserting* items one by one into a list that is already in the correct order. It is not often that we need to use such a procedure with numbers. More often we start with a complete but jumbled set of numbers which we want to sort into order. There are several ways of doing this on a computer. The program in Listing 8.4 shows the simplest way.

```
10 PRINT"□"
20 INPUT"HOW MANY";N
30 DIM A(N)
40 FOR J = 1 TO N
50 INPUTA(J)
60 NEXT J
70 FOR J = 1 TO N-1
80 FOR K = J + 1 TO N
90 IF A(K) >= A(J) THEN GOTO 130
100 T = A(K)
110 A(K) = A(J)
120 A(J) = T
130 NEXT K
140 NEXT J
150 PRINT"□"
160 FOR J = 1 TO N
170 PRINTA(J)
180 NEXT J
```

Listing 8.4. Sorting numbers by the interchange sort method.

The program begins by asking how many numbers are to be sorted, then asks the user to type them in one at a time. They can, of course, be typed in any order. They are stored in array A.

Figure 8.3 shows how this method of sorting works. It has two nested loops. The first time round, when J is 1, the first number in the array is compared with all those following it, using the inner loop which calls up all the numbers from A(K), to A(N). In the expression A(K), K has the value J + 1 so when J = 1, the first number called for is N(2). On the second time round the outer loop the first number called for is A(3), and so on for successive times round the outer loop. The numbers are compared with the first number at line 9ϕ. If A(K) is greater than or less equal to the first number, nothing happens; the next A(K) is taken for comparison. But if A(K) is less than the first number, the two numbers are interchanged.

Interchange requires a variable T, for temporary storage. Figure 8.4 shows how lines 1ϕϕ to 12ϕ work.

Having found the smallest of all the numbers and put it first in the

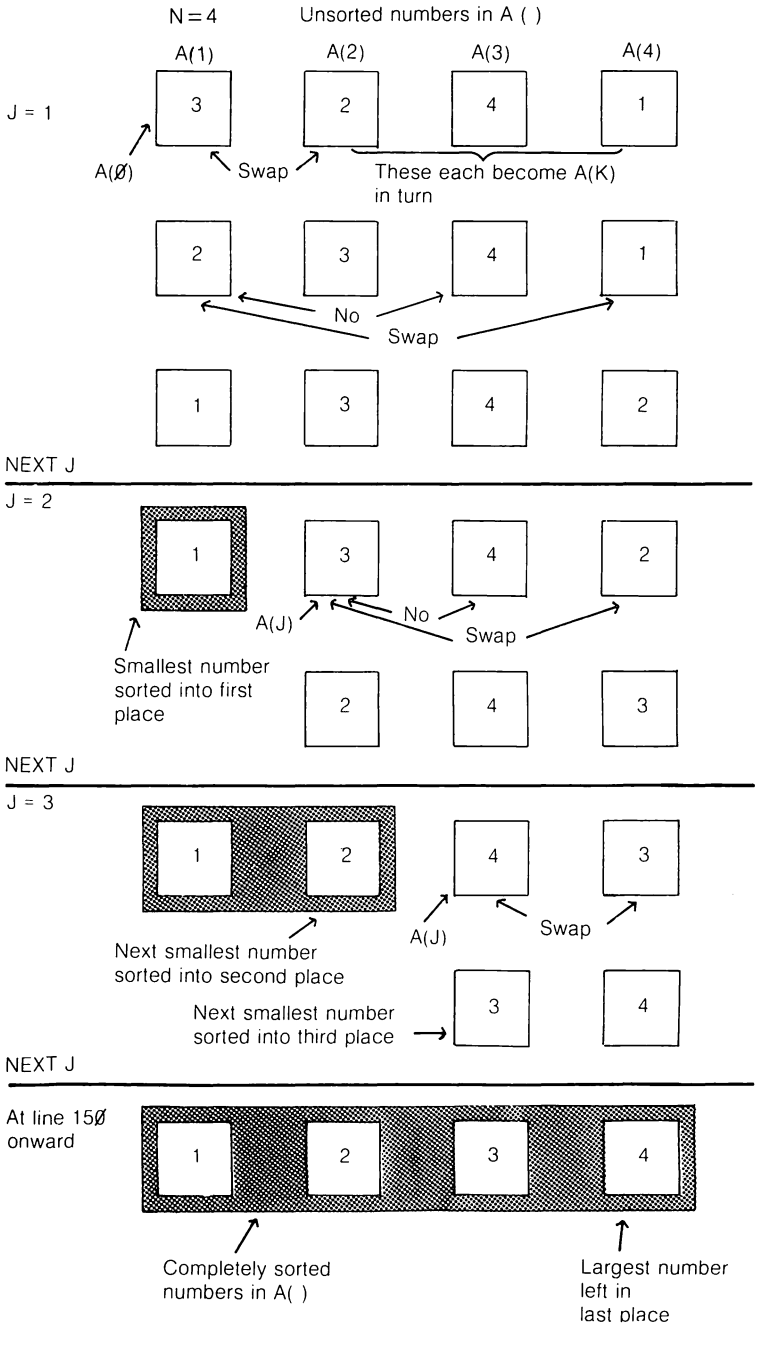


Fig. 8.3. How the interchange sort works, showing the contents of A( ) after every swap. 'No' means 'no swap'.

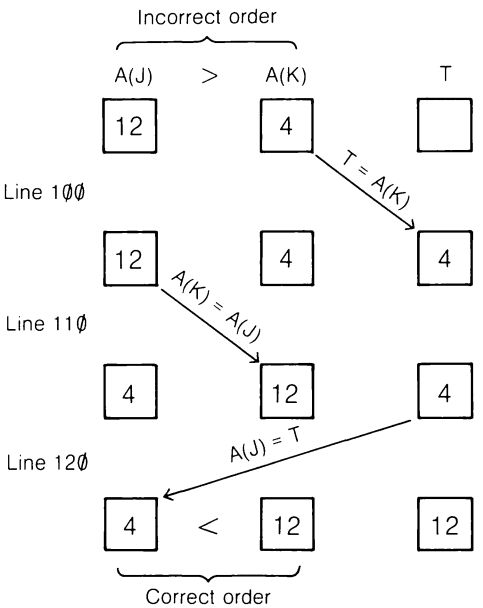


Fig. 8.4. Using a temporary variable T for swapping two numbers between A(J) and A(K) in the interchange and bubble sorting programs.

array, the micro ignores this number from now on. The outer loop repeats with J equal to 2. Now the second number is compared with all those that follow it (A(3) onwards) and, as before, the smallest of those which remain is put at second place. On the third time round the smallest of the remaining numbers (A(4) onwards) is put at third place. Eventually the largest number of all is left at the Nth place. There is no need to repeat the loop N times, for the Nth number will already have been correctly put in its place at the (N − 1)th repetition of the outer loop.

RUN this program several times with different numbers of entries to sort. It works quickly enough when there are only a half-dozen or so numbers but, with twenty numbers it takes an appreciable time. The problem is that the computer has to take each number in turn and compare it with all the numbers listed after it. The more numbers, the more following numbers there are to take and the more following numbers to compare it with. It takes four times as long to sort twice as many numbers.

You can look more closely into how the program works by adding these extra lines:

```
125 FOR L = 1 TO N:PRINT A(L);:NEXT L
126 PRINT" "
135 PRINT"*";
```

Note the semicolons in lines 125 and 135. Delete line 15Ø. Now the display shows the result of every interchange. It prints a '*' each time it reaches the end of the outer loop. If you see two or more asterisks together, it indicates that there was no interchange during that outer loop. The micro has been wasting time by inspecting numbers which are already in the correct order. This simple sorting program can also be used for sorting words. Just amend the variables A() to A$() and T() to T$().

Bubble sort

This program is a little more complicated than the interchange sort but it can be much quicker. The way it works is shown in Fig. 8.5. The computer works down the array of numbers taking them in pairs. Whenever it finds a pair that are in the wrong order, it swaps them round. Then it goes back and does the same thing all over again. Every time it begins working down the list it sets a flag, F, to Ø. If it makes a swap, it sets F to 1. If it works through the list and no swaps are found necessary, the numbers must then all be in the correct order, so sorting is complete. If the numbers are reasonably well in order to begin with, they are soon sorted. Once the job is done the flag tells the computer that this is so and no further time is wasted with unnecessary operations. Twenty numbers more-or-less in the correct order are sorted extremely quickly. On the other hand, if they are in an almost reverse order the bubble sort takes longer than the interchange sort described earlier. In the bubble sort the smaller numbers find their way to the top of the list gradually, like bubble rising through a liquid. This is the feature which gives this method its name.

As in the interchange program, two values are swapped by using a temporary variable (Fig. 8.4). The three lines which do this (lines 1ØØ to 12Ø) could be put on to one multistatement line. This would make the program RUN a little faster. Bubble sort can be adapted to sorting words in the same way as the interchange program.

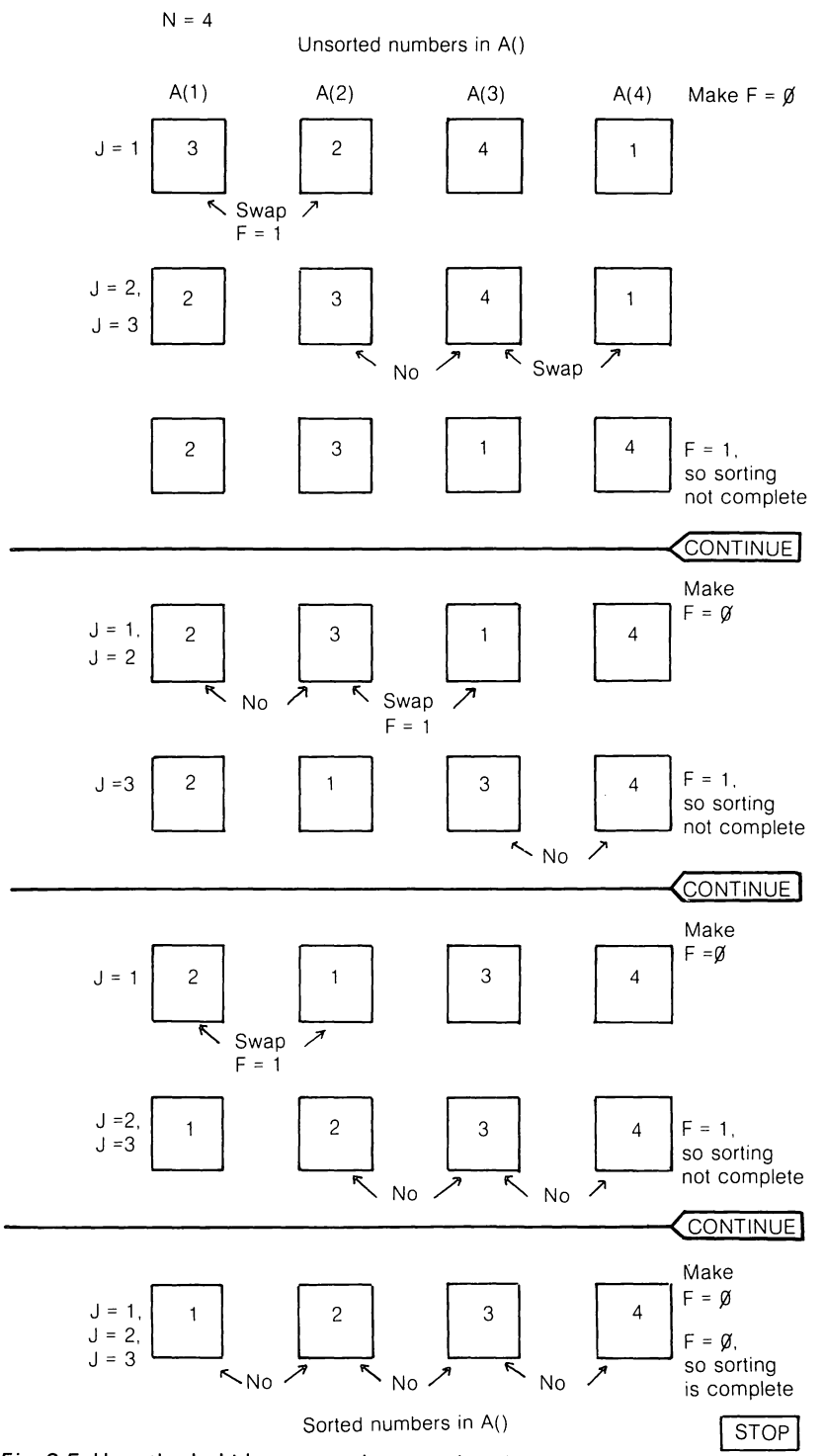


Fig. 8.5. How the bubble sort works, showing the contents of A() after every swap. 'No' means 'no swap'.

```
10 PRINT"□"
20 INPUT"HOW MANY";N
30 DIM A(N)
40 FOR J = 1 TO N
50 INPUT A(J)
60 NEXT J
70 J = 1
80 F = 0
90 IF A(J) <= A(J+1) THEN GOTO 130
100 T = A(J)
110 A(J) = A(J+1)
120 A(J+1) = T
125 F = 1
130 J = J + 1: IF J < N THEN GOTO 90
140 IF F = 1 THEN GOTO 70
150 PRINT"□"
160 FOR J = 1 TO N
170 PRINT A(J)
180 NEXT J
```

Listing 8.5. Sorting numbers by the bubble sort method.

To summarise

In this chapter you have found out how to:

- use the LEFT$, RIGHT$ and MID$ functions to manipulate portions of strings
- use the STR$ and VAL functions to convert numbers to strings and strings to numbers
- use the LEN and FRE(∅) functions
- sort words into alphabetical order, using the relational operators < and >
- sort words or numbers using the interchange sorting method
- sort words or numbers using the bubble sorting method.

Chapter Nine

More About Sound

The details of the VIC's sound generators and how to handle them have already been dealt with in Chapter Five. This chapter shows you a quick way of creating special sound effects to add to your programs.

One of the most difficult things about sound effects is that it is almost impossible to predict how they will sound when the program is run. For example, if you want the sound of a police siren, exactly which number should be POKEd into which sound generator to begin with? Which value should it be made to rise to? How long should it take to rise? These questions can only be answered satisfactorily by trying various sets of values and listening to the result. Then it may be obvious that one or more of the values badly needs adjusting. But the other values do not need to be changed. The only trouble is that you still have the bother of entering them all over again.

The FXG (short for Effects Generator) program (see Listing 9.1) allows you to try a sound effect repeatedly, adjusting one or more of the values, but leaving the others unchanged. The screen shows a table of all the values that are in use, so that you can clearly see what you are doing. When the sound is right, you have only to copy the values from the screen and insert them in your program.

Like the character generator programs of Chapter Seven, FXG is a rather complicated program in itself. This is inevitable, for it takes a large amount of the work out of your hands and makes the micro do it instead. If you simply want to invent your own sound effects, but have little interest in how the program works, turn to the section 'Using FXG', where instructions for using it are set out. For those who are 'into' programming in a serious way, here is a brief description of how the program works.

Line 1Ø sets R to the base address of the VIC sound generators. C$ is used in lines 2Ø to 4Ø, which PRINT the display (Fig. 9.1). After

```
10 R = 36874:PRINT"■":PRINT:C$ = "DDLLLM
MMHHHNNNVV"
20 FOR J = 1 TO 5
30 PRINTTAB(2)J;C$:PRINTTAB(2)J;"**";RIG
HT$(C$,14)
40 NEXT J
50 PRINTTAB(5)C$:PRINT"■"
60 INPUTN$(1),N$(2),N$(3),N$(4),N$(5),N$
(6),N$(7),N$(8),N$(9),N$(10)
70 FOR J = 1 TO 10
80 A(J,0) = VAL(MID$(N$(J),3,2))
90 A(J,1) = VAL(MID$(N$(J),5,3))
100 A(J,2) = VAL(MID$(N$(J),8,3))
110 A(J,3) = VAL(MID$(N$(J),11,3))
120 A(J,4) = VAL(MID$(N$(J),14,3))
130 A(J,5) = VAL(RIGHT$(N$(J),2))
140 NEXT J
150 FOR J = 1 TO 9 STEP 2
160 D = A(J,0):IF D = 0 THEN GOTO 220
170 FOR K = 1 TO D
180 FOR L = 1 TO 5
190 POKER+L-1,A(J,L)+(A(J+1,L)-A(J,L))*K
/D
200 NEXT L
210 NEXT K
220 NEXT J
230 FOR L = 0 TO 4:POKE R + L, 0:NEXT L
240 PRINT"■"
250 GOTO 60
```

Listing 9.1. FXG, a sound effects utility program which will help you to design sounds to go with your programs.

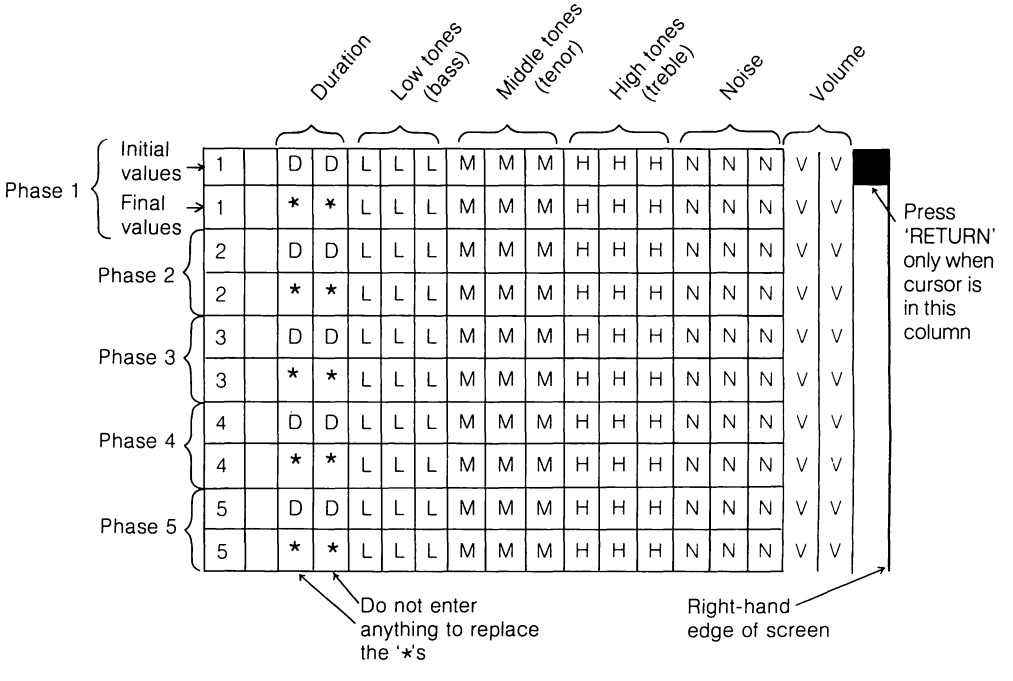


Fig. 9.1. The display of the FXG program. The rows of letters show you where to type in the values for the sound effects.

the display has been printed, line 5ø sends the cursor 'home'. Now the INPUT in line 6ø prints a question mark near the top of the screen just above the display. The computer is now waiting for the data to be typed in.

One of the less usual features of the VIC is that it takes INPUT from the screen. Most other microcomputers have a part of memory called the *input buffer*. Anything which is being typed in to the machine goes into this buffer first. It may or may not also be displayed on the screen. When the user has typed in text in response to the INPUT command, and has pressed RETURN, most other micros go to the input buffer to find out what has been typed in. The VIC goes to the screen RAM instead and finds out what is being displayed on the screen. This feature of the VIC is made full use of in the FXG program. The cursor is always sent back home immediately before an INPUT command. You then move it over the characters displayed on the screen using the left-right cursor key. You press RETURN at the end of each line of the display. The characters over which the cursor has just been moved are then read as INPUT. The advantage of this technique is that when you want to alter just a few details of the sound effects, all you have to do is to change a few of the figures in the display, by retyping them. Those figures which are to remain unaltered from the previous trial are still there on the screen, and are read in as INPUT simply by scanning over them with the cursor. Lines in which no alterations are required are simple scanned with the cursor, and RETURN is then pressed. With so many figures involved, yet so few to alter, this saves much time and is a great convenience.

The 10 rows of the display are read separately into the array N$(). In lines 8ø to 13ø, segments from each item in N$ are turned into numerical values (using VAL) and put into another array (A). The contents of A() are illustrated in Fig. 9.2.

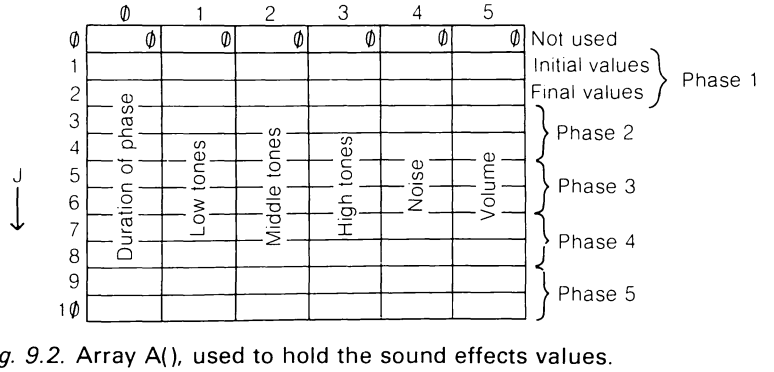


Fig. 9.2. Array A(), used to hold the sound effects values.

The program then steps through A(), reading the odd-numbered lines to find out what initial values are to be POKEd to the sound generators at the beginning of each phase of the sound. The even-numbered lines of A() hold the final values. Line 16ϕ finds out if the duration of the sound is to be zero, in other words, if a phase is to be skipped. Lines 18ϕ to 2ϕϕ, POKE the generators and volume control in turn with values taken from A(). The value to be POKEd at any stage depends on the initial value, the final value, the final value that K will reach in line 17ϕ (given by D), and the current value of K. From Fig. 9.3, it is clear that the formula required is:

$$\text{VALUE} = \text{INITIAL} + \frac{(\text{FINAL} - \text{INITIAL})*\text{K}}{\text{D}}$$

in which K is the current value of K each time round the loop. This calculation is done for each set of values from A(), in line 19ϕ.

When the sound effect is complete, all generators and the volume control are set to zero in line 23ϕ. The cursor is then sent back to 'home' and the program jumps back to line 6ϕ to receive fresh INPUT.

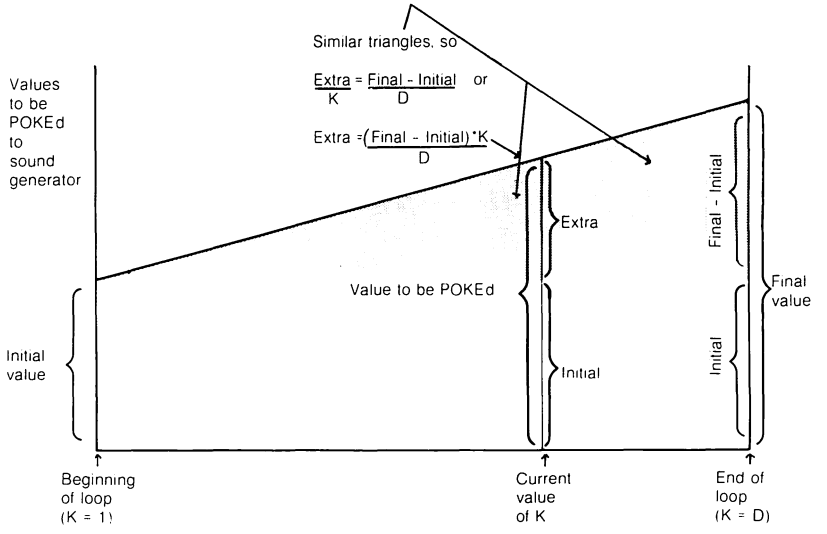

Similar triangles, so
Extra / K = Final - Initial / D or
Extra = (Final - Initial)*K / D
Values to be POKEd to sound generator
Extra
Final - Initial
Value to be POKEd
Final value
Initial value
Initial
Initial
Beginning of loop (K = 1)
Current value of K
End of loop (K = D)

Fig. 9.3. How the values for rising and falling tones are calculated by the computer.

Using FXG

The idea of this program is that it makes the computer produce a sound effect which may have up to five phases. As an example, take the sound effect used in the Air Raid program (Listing 5.4). Phase 1 was the descending whine of the falling bomb, gradually increasing in volume. Phase 2 used only the noise generator at full volume to produce the explosion. In phase 3 the noise died away and was replaced in phase 4 by the low drone of the aeroplane engines. For each phase, you have to decide on what length of time the phase is to last, which generators are to be used, what values are to be POKEd to start with (initial values) and what values are required at the end of the phase (final values). You also have to decide how the volume of sound is to alter during each phase. These are the stages of using the program:

(1) RUN the program. The display appears as in Fig. 9.1, with a question mark above it, on the left.
(2) Press the 'up-down' cursor key *once* to bring the cursor down to the first line of the display.
(3) Move the cursor over the '1' at the beginning of the first line, using the left-right cursor key, and move toward the right, until it is on the first of the two 'D's. You have to type the duration of the first phase here. It must be a two-digit number. Try '1Ø' to begin with. This gives a phase just over 1 second long. If you want to make the phase shorter, you can use a number less than 1Ø, but type it with an initial zero (e.g. 'Ø7'). If you make a mistake, move the cursor back, by pressing SHIFT with the left-right cursor key until the cursor is on the incorrect figure. Retype the figure correctly.
(4) Carry on along the line typing figures in place of:

LLL – for lower tone generator (bass)
MMM – for middle tone generator (tenor)
HHH – for high tone generator (treble)
NNN – for noise generator

You do not need to use *all* of these, in fact, you need only use one (say, the middle tone generator) and leave the LLL, HHH, and NNN unaltered. Figures must be in the proper range, between 128 and 254. Finally move the cursor over VV, and type the volume required (Ø1 to 15).
(5) The cursor is now on the blank space at the right-hand end of the first line (Fig. 9.1). Press RETURN, which makes the cursor jump to the second line. It appears on top of the '1'. Two question marks ('??') appear on the left.

(6) Move the cursor past the '**' at the beginning of the line. Now type in the final values for those generators which you have given initial values to in the first line. As before, there is no need to type in values for generators which you are not going to use. Press RETURN when the cursor reaches the blank space at the end of the line.
(7) If the sound effect is to have more than one phase, continue as in steps (3) to (6) above for the lines beginning with '2', '3', '4' and '5' for as many phases as you want. The lowest line of letters (with no number) is there just to help you keep your bearings. Remember that it is essential to press RETURN when the cursor reaches the blank space at the end of *every* line.
(8) When you have entered details of all the phases you require, skip to the end of the display by pressing RETURN repeatedly. There is no need to move the cursor along the unused lines. After pressing RETURN for the second '5' line, there is a short pause, then the sound effect is heard.
(9) After the effect is finished, the cursor reappears at the top left corner of the screen. To repeat the effect, just press RETURN.
(10) To revise the effect, move the cursor over the display, retyping any figures you want to alter, and typing any new figures you want to add. Press RETURN at the end of each line, just as in steps (3) to (7) above. Numbers which are not altered are carried through to the new effect.
(11) When you are satisfied with the effect, copy the values from the screen.

The main limitation of the FXG program is with rising or falling tones of higher pitch. This is partly due to the way the generators calculate frequency. At the upper end of the scale an increase or decrease of 2 or 3 in the value POKEd to a tone generator produces an interval of a semitone or even a whole tone. Instead of a smoothly gliding note (such as we get with the lower notes) we get a series of distinct notes of rising or falling pitch. One of the reasons for this is that the calculations required during the running of the program force any changes of pitch to be made in a few large steps rather than in many small ones. If you are wanting to control two or three sound generators at once, this is hard to avoid but, if your sound effect requires only one generator to be controlled at a time, it is possible to improve upon the effect by making the steps smaller and having more of them. This point is dealt with in the example below.

Here is an example of turning figures obtained from FXG into lines to be inserted in one of your programs. The effect is that of a

motorcycle roaring along a road, skidding to a halt with the sounds of breaking glass, and then crashing. Not very pleasant in real life, but harmless enough as a sound effect! It is divided into four phases:
(1) A low note of increasing pitch from the bass generator. The screen display is shown in Listing 9.2.

```
1  30150MMMHHHNNN15
1  **180MMMHHHNNN15
```

Listing 9.2. The screen display of FXG for the first phase of an effect, the motorcycle.

The phase lasts about three seconds, and is at full volume.
(2) A rising high-pitched note combined with a high-pitched noise, to give the sound of breaking glass with skidding tyres. This is a short phase, lasting a little less than one second. The display is shown in Listing 9.3.

```
2  08LLLMMM25025415
2  **LLLMMM25225415
```

Listing 9.3. The screen display of FXG for the second phase of an effect, the skid.

As before, the volume is at its maximum.
(3) A crashing sound at full volume is shown in Listing 9.4.

```
3  06LLLMMMHHH20015
3  **LLLMMMHHH22015
```

Listing 9.4. The screen display of FXG for the third phase of an effect, the crash.

Note the rising pitch of the noise. There is plenty of scope for playing around with values to get precisely the effect you want.
(4) The crashing noise dying away is shown in Listing 9.5.

```
4  15LLLMMMHHH22012
4  **LLLMMMHHH22000
```

Listing 9.5. The screen display of FXG for the fourth phase of an effect, the dying echoes of the crash.

The final volume is zero and this takes just over 1.5 seconds to happen.

Try RUNning the FXG program and setting it to give this effect. Alter a few of the values to improve it; you may have different ideas about timing and pitch.

When used in a program, this effect is written as four separate loops, one for each phase. It makes the program easier to understand if we define the addresses of the sound generators and volume control as 4 variables (see Listing 9.6).

The tenor generator is not being used, so its address is not defined.

```
10 S1 = 36874:S3 = 36876:N = 36877:V = 3
6878
```

Listing 9.6. Setting up the variables for the crash effect.

Phase 1 is a loop lasting about three seconds. Its essential structure is illustrated in Listing 9.7.

```
20 POKE V,15
30 FOR J = 1 TO X
40 POKE S1, 150 + 30*J/X
50 NEXT J
```

Listing 9.7. Trial program for phase 1.

The calculation of the value in line 4ø uses the equation shown earlier in this chapter. The initial value is 15ø and the final value is 18ø; their difference gives the figure '30', which is then multiplied by J and divided by X (the number of steps). Before testing this, type in the line shown in Listing 9.8 to silence everything at the end.

```
200 POKE S1,0:POKE S3,0:POKE N,0:POKE V,
0
```

Listing 9.8. Silencing the generators at the end of the program.

Give X a value by typing in an extra line which is removed later (see Listing 9.9).

```
25 X = 200
```

Listing 9.9. A trial value for the loop.

Now RUN the program. You hear the sound of the motorcycle. This will be the same as that you heard with the FXG program, but you may think that the phase does not last long enough. Since there is only one address to be POKEd each time around the loop, the loop is running much more quickly than the loop in FXG. To make it take longer, alter line 25 as shown in Listing 9.10. Try again now,

```
25 X = 300
```

Listing 9.10. Another trial value.

and you will probably agree that this sounds right. As a rough guide, you can assume that a count to 100 takes about one second when there is only one address to be POKEd. A count to 300 should take the required three seconds. Now that 300 is established as a suitable value for X, this can be written into the program, and line 25 is deleted. The amended loop is shown in listing 9.11. Note how '30

```
30 FOR J = 1 TO 300
40 POKE S1, 150 + J/10
50 NEXT J
```

Listing 9.11. The final version of the loop for the first phase.

*J/300' is simplified to 'J/10'. Now to program phase 2, the skid and breaking glass. This can have much the same form as the loop in Listing 9.11, but uses the treble generator instead of the bass. Before the loop begins, the noise generator is POKEd to give the screech. The volume control is already at 15, so it does not need to be altered. One important point - remember to silence the bass generator!

```
60 POKE S1,0:POKE N,254
70 FOR J = 1 TO X
80 POKE S3,250 + 2*J/X
90 NEXT J
```

Listing 9.12. Trial program for phase 2.

As before, the formula shown earlier in this chapter has been used for calculating the value to be POKEd. We type in an extra line to allow different values of X to be tried. Since this phase is to last for a little less than one second, the line might be as in Listing 9.13. This

```
65 X = 80
```

Listing 9.13. A trial value for the loop of phase 2.

gives a sound of about the right duration, so line 65 may now be deleted and lines 70 and 80 amended as shown in Listing 9.14. Again '2*J/80' is simplified to 'J/40'

```
70 FOR J = 1 TO 80
80 POKE S3,250 + J/40
90 NEXT J
```

Listing 9.14. The final version for phase 2.

Phase 3 needs the noise generator to increase in pitch while the volume falls slightly, from 15 to 12. Before the loop begins, the treble generator must be silenced (see Listing 9.15).

```
100 POKE S3,0
110 FOR J = 1 TO X
120 POKE N,220 + 20*J/X
130 POKE V,15 - 3*J/X
140 NEXT J
```

Listing 9.15. Trial program for phase 3.

Line 105 is used to find a good value for X. The value 60 is a little too high, for now there are two registers to be POKEd in the loop. A value of 30 or 40 seems to be about right. Try this by typing in:

```
105 X = 30
```

Listing 9.16. A trial value for the loop of phase 3.

If this sounds right, alter lines 110 to 140 as shown in Listing 9.17. Once again, the values to be POKEd are found by substituting the selected value of X in the expressions, cancelling where necessary.

```
110 FOR J = 1 TO 30
120 POKE N,220 + 2*J/3
130 POKE V,15 - J/10
140 NEXT J
```

Listing 9.17. The final version for phase 3.

The final phase maintains the noise generator at constant pitch, but volume decreases to zero. The previous phase leaves it already set to 12 at the beginning of this phase and the noise generator is already set to 22ø, so programming is simple (see Listing 9.18). This

```
150 FOR J = 1 TO X
160 POKE V,12 - 12*J/X
170 NEXT J
```

Listing 9.18. Trial program for phase 4.

phase lasts about 1.5 seconds and, since there is only one value to be POKEd each time round the loop, we can assume that 15ø is likely to be an acceptable value for X (see Listing 9.19).

```
145 X = 150
```

Listing 9.19. A trial value for the loop of phase 4.

This turns out to be very suitable, so the lines are amended as shown in Listing 9.20.

```
150 FOR J = 1 TO 150
160 POKE V,12 - 2*J/25
170 NEXT J
```

Listing 9.20. The final version for phase 4.

The original effect created by FXG has now been converted into lines which may be inserted into any BASIC program at the point where the sound effect is required. If it is required several times, it is best made into a subroutine. It must then end with RETURN.

This example has shown how an effect designed with the aid of FXG can be incorporated into your programs. FXG is a useful program, but has been made as short as possible, so that it does not take too long to type in. When you have become familiar with using it, you may like to add improvements of your own. One helpful addition would be to allow the user to make the sound repeat indefinitely. this would make it easier to create repetitive effects, such as the sound of machinery, locomotives, clucking birds, sirens and the like. You could also increase the number of phases that can be included. Using FXG will help you set off into the ear-splitting world of VIC sound effects. Having designed your effects using this program, you will find it easy to make subtle changes to perfect them.

To summarise

In this chapter, you have found out:

- how to design novel sound effects, using the FXG utility program
- how to incorporate your sound effects into a BASIC program.

Chapter Ten
The Functioning Machine

Assuming that your VIC really is functioning, in the sense of behaving in the way that is intended, let us look at some of its functions. The word 'function' is one that is over-used in computing literature. As you can see, 'function' and its related forms have been used five times already in this chapter! It is a word with a wide meaning, covering almost anything involving 'doing something' so it has been easy for computer people to press it into service in a number of virtually meaningless and unrelated ways.

Function keys

These are the four light brown keys on the right-hand side of the keyboard. Their function (sorry, their job) is to allow you to interact with a program when it is running. These keys are used by pressing one of them when the computer is obeying the GET$ statement. The RETURN key does not have to be pressed.

The function keys are particularly handy for games. They can be used for firing rockets, controlling the motion of a spacecraft, or of an adventurer in a maze. They have uses in programs of other kinds too. For example, in a word processing program they can be used for issuing special commands concerned with text formatting.

Listing 10.1 is a program which shows how the function keys may be employed.

When the program is RUN, an aeroplane flies across the screen many times. The user has to press a function key to shoot the aeroplane down. It can be shot down only when it is exactly in the centre of the screen. Most of the programming features have already been described in earlier chapters, so we will concentrate on how the function keys are being used. However, there is one new function (that word again!) in this program which has not been described

```
10 N = 1
20 PRINT"■■■":POKE 36879,59
30 GET A$: IF A$ <> CHR$(133) THEN GOTO
30
40 FOR J = 1 TO 110
50 PRINTTAB(J)"■■■";
60 GET A$
70 IF A$ = CHR$(136) AND POS(0) = 12 THE
N GOTO 110
80 PRINT"■"
90 NEXT J
100 N = N + 1: GOTO 40
110 PRINT"■":POKE 36879,43:M = M + 1
120 FOR K = T TO 100: NEXT K
130 POKE 36879,59:PRINT"■■LIGHTS";N;"
HITS";M
140 GET A$: IF A$ <> CHR$(134) THEN GOTO
 140
150 GOTO 20
```

Listing 10.1. Target, a game to show how the function keys are used.

before. This is POS, which appears in line 7Ø. It has an argument 'Ø', but this is a dummy and it does not matter what value it has, for it is meaningless. POS takes the value of the column of the screen on which the next character to be printed will appear. In this program we use this to detect how far the aeroplane has travelled on its flight across the screen. When its nose gets to column 11, the next PRINTING position is column 12 and POS(Ø) gives the value 12. When it is in this position, the aeroplane may be shot down.

Now to consider the function keys. Line 3Ø has the statement GET A$. When the computer reaches this line, it scans the keyboard to see what key (if any) is being pressed at that moment. If no key is being pressed, A$ becomes " " (an empty string). If a key is being pressed, A$ becomes a single-character string, holding the character marked on that key. Line 3Ø compares A$ with the ASCII code for a character by using the function CHR$. Although the function keys do not cause any visible character to be printed on the screen, they each have a pair of ASCII codes (unshifted and shifted), as listed in Table 11 in Appendix A. Since they are not standard ASCII codes, it is better to refer to them as CHR$ codes. The code for function key F1, unshifted, is 133. The computer repeats line 3Ø indefinitely, until key F1 is pressed (unshifted). The result is that, when the program is run, the screen remains a clear turquoise blue (cyan) until key F1 is pressed. Then the aeroplane starts to fly. It flies from left to right and, after leaving the screen on the right, reappears on the left, one line lower. At the end of this 'flight', it repeats, beginning from the top of the screen once again. The number of 'flights' is recorded by variable N.

At line 7Ø the keyboard is scanned once more. This time the

computer is checking to see if key F7 is being pressed (CHR$ code = 136). If the player presses F7 at *exactly* the same time as the aeroplane reaches column 11, the plane is shot down.

The program then jumps to line 11Ø where the screen is flashed red. Then the score is displayed. Line 14Ø holds the display until key F3 is pressed, when the program begins again.

As far as the player is concerned, the sequence is:

(1) RUN the program.
(2) When you are ready to play, press F1.
(3) To fire at the aeroplane, press F7. Aim to do this just as the aeroplane reaches the middle of the screen.
(4) To clear the scores from the screen, ready to play again, press F3.

Here is another chance for the sound effects enthusiast; add the droning sound of the aeroplane's engines, and the explosion when the aeroplane is hit.

BASIC functions

Many of the words used in BASIC programs are called functions. Again the emphasis is on 'doing'. A function takes a variable and 'does something' with it. Usually the variable is placed in brackets after the word and is called the *argument* of the function. Chapter Eight described the action of the string functions, which 'do things' to strings. There are also many numeric functions, such as INT and RND. INT, for example, takes the number or variable used as its argument and removes all figures after the decimal point.

Another useful BASIC function is SQR, which returns the square root of a number. Try this:

```
1Ø INPUT X
2Ø PRINT SQR(X)
```

In an instant, it works out the square root of X to as many as nine significant figures. If you INPUT a negative value, the micro prints an error message "? ILLEGAL QUANTITY ERROR IN 2Ø", for the square root of a negative number is an imaginary quantity which the VIC is not able to understand.

Several of the functions are concerned with angles. The ones available on the VIC are:

SIN(X) which gives the sine of the angle X.

COS(X) which gives the cosine of the angle X.
TAN(X) which gives the tangent of the angle X.
ATN(X) which gives the angle which has the tangent X.

TAN and ATN are the reverse of each other. In all these functions, the angles are measured in radians, not in the degrees to which most people are accustomed. So, if you are using these in a program and want the user to be able to enter angles in degrees, or you want to print out the answers in degrees, you must remember to convert from degrees to radians or from radians to degrees at the appropriate stage. An easy way to do this is described later in this section.

Listing 10.2 shows a program to PRINT a circle (of sorts). Since the screen characters are rectangular in shape, it draws an ellipse rather than a true circle. However, it illustrates the principles of drawing circles, which can be useful in certain types of program, especially those involving orbiting spacecraft and satellites. It also shows how data regarding such objects may be calculated in a program, even if it is not used as the basis for a display. This program allows you to INPUT the required radius (∅ to 1∅, and it need not be an integer).

```
10 PRINT"□"
20 INPUT R
25 PRINT"□"
30 FOR J = 0 TO 2*π STEP .3
40 FOR K = 1 TO 10 - R*COS(J)
50 PRINT" "
60 NEXT K
80 PRINTTAB(11+R*SIN(J))"●";
85 PRINT"■"
90 NEXT J
```

Listing 10.2. This program draws circles of any required radius.

Before we go on to examine the calculations there is something to be noticed in line 3∅. The character 'π' is typed in by using the SHIFT key with the up-arrow key. The odd thing about π is that it is not just another character or graphics symbol like the hearts, clubs and other designs we find on the fronts of the keys. It is a variable, with a fixed value. Its value is the very special one, 3.14159265, that special number which relates the radius of a circle to its circumference. As it happens, there are 2 π radians in 360° so, if J is made to vary from 0 to 2* π, it represents an angle J increasing from 0° to 360° – a complete revolution. This is a convenient way of working, for we do not have to convert degrees into radians before evaluating sines and cosines in lines 4∅ and 8∅.

Figure 10.1 shows the way we calculate where to plot the disc-like

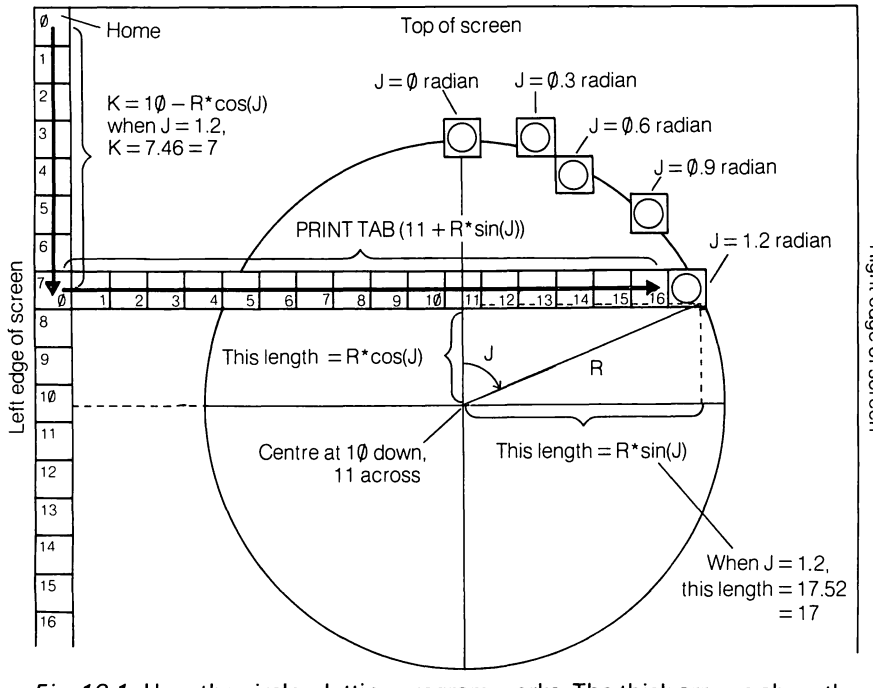


Fig. 10.1. How the circle-plotting program works. The thick arrows show the motion of the cursor before a disc is printed.

graphics characters which form the circumference of the circle. The centre of the circle is located on the tenth row and eleventh column of the screen. To plot each disc the cursor has to move down the screen a certain number of rows, then move along a certain number of columns. Then a disc is PRINTed. The number of rows down the screen is determined by the loop in lines 4Ø to 6Ø. The loop runs K times and each time round it PRINTS " ", so making the cursor move down by one line. The maximum value of K for each point on the circumference is calculated as:

1Ø – R*COS(J)

Figure 10.1 shows how this expression is derived. Having arrived at the correct row, the cursor is moved along the row by using the TAB function. The distance to be moved is calculated as:

11 + R*SIN(J)

The values calculated by the expression as above are exact and could be stored in memory for later use. In a program which simulates orbiting space flight, they give the co-ordinates in space of the spacecraft with nine-figure accuracy. But before they are used for

plotting the discs in this program, they are automatically rounded to the nearest integer. It is not possible for the VIC to PRINT a disc, exactly 4.53458761 lines down and 13.6543874 columns across! With the limited number of rows and columns on the VIC screen the circle is unavoidably a rough one.

While on the subject of circles, Listing 10.3 shows a program which tells you everything you could possibly want to know about circles and spheres, given their radius.

```
10 PRINT"□"
20 INPUT "RADIUS";R
30 PRINT"CIRCUMFERENCE =";2*π*R
40 PRINT"AREA =";π*R↑2
50 PRINT"VOLUME OF SPHERE =";4*π*R↑3/3
60 PRINT"AREA OF SPHERE =";4*π*R↑2
```

Listing 10.3. Making full use of 'π'.

This uses the variable 'π' in a number of standard calculations. As you can verify by running the program, the value of 'π' does not have to be declared at the beginning. Its value is already fixed within the computer. Just to show that 'π' can also act as an ordinary character, this program is called Using π.

Another use for 'π' is in converting between degrees and radians. Since 180° equal *exactly* π radians, the conversion is simple:

```
1Ø INPUT"DEGREES";D
2Ø PRINT"EQUALS";D*π/18Ø;"RADIANS"
```

Or the reverse calculation:

```
1Ø INPUT"RADIANS";R
2Ø PRINT"EQUALS";R*18Ø/π;"DEGREES"
```

The expressions above may be used in any program for performing these conversions.

The function function

As explained in the previous section, a BASIC function takes a number and 'does something' with it. VIC BASIC provides a useful assortment of functions of many kinds, most of which we have used in one way or another in this book.

Occasionally you need a function that is not among those provided. If so, you are free to write your own. For example, you may need a function which calculates the volume of a sphere, given its radius. If you need this only once or twice in a program it is no

trouble to use the expression 'V = 4*π*R↑3/3' whenever the volume has to be calculated. But if this calculation has to be done many times it is useful if the computer is told once and for all how to calculate the volume. Then it is simply given the radius each time and uses its stored information about how to do the calculation. This is what the function function does.

Before such a user-defined function can be used it must be defined. There is a DEF (define) statement for this purpose. This is put on a program line and, since the function must be defined before it can be used, it is best put near the beginning of the program:

```
1Ø DEF FNV(R) = 4*π*R↑3/3
2Ø INPUT R
3Ø PRINT FNV(R)
```

Line 1Ø defines the function V (for volume) which calculates the volume of a sphere, given its radius R. The name of this function is V, but it can have any other name of the kind that is given to variables. Normally we use one or two letters or a letter followed by a figure, but it is quite in order to define FNVOLUME, or even FNJIM! The argument of the function (R) is the value which the function is to 'do something' with.

In the example above, the variable R was used in the definition of the function and was used again to INPUT a trial value and to call the function at line 3Ø. There is no need to use the same variable name all the way through. When the function is defined, the variable which is used in the argument must, of course, also be used in the definition. In effect what we are saying is: 'Take a value, call it R, then cube it, multiply it by "pi" and by 4, divide by 3 and tell us the answer'. But when using this function, the variable upon which the function is to operate can have any name. You could alter lines 2Ø and 3Ø of the program to:

```
2Ø INPUT T
3Ø PRINT FNV(T)
```

The micro will now take the value of T, and do the same things with it as it was told to do with R. You can also have a numeric expression as the argument. For example, change line 3Ø to:

```
3Ø PRINT FNV(2*T+7)
```

This doubles T, adds 7 to it and *then* sends it to the function. The answer is the volume of a sphere with a radius of 7 plus twice the value T.

Functions of functions

When you ran the Using π program, you may have marvelled at all the figures it produces. It tells you, for example, that a sphere of radius 4 cm has a volume of 268.082573 cm. One of the reasons for such a long answer is that the value used for π has nine significant figures. In practice, such long numbers are almost meaningless. If the radius of the sphere is measured only to the nearest millimetre (0.1 cm) it is nonsense to calculate the volume to the nearest millionth of a cubic centimetre. It would be more sensible to round off the answer to the nearest whole number, or integer. The INT function does not do this properly, for it simply chops off the figures after the decimal place. INT(5.9) comes out as 5. If we round off 5.9 in the *usual* way, the answer we expect is 6. This is because numbers below the halfway mark (less than 5.5, for example) are rounded down (to 5) while numbers at the halfway mark or above (equal to or greater than 5.5) are rounded up (to 6).

Rounding to the nearest integer is simple. If 0.5 is added to the number before INT is used, the number is rounded correctly:

```
10 INPUT N
20 PRINT INT(N + .5)
```

Try this with several numbers, such as 5.3, 5.5, 5.7, −3.234, or any others you like. It always rounds correctly, even with negative numbers. We can make this into a function R (for rounding), which is defined as follows:

```
10 DEF FNR(N) = INT(N + .5)
```

Now we can add this function to the Using π program to make it print out all the results to the nearest whole number (see Listing 10.4).

```
5 DEF FNR(N) = INT(N+.5)
10 PRINT"□"
20 INPUT "RADIUS";R
30 PRINT"CIRCUMFERENCE =";FNR(2*π*R)
40 PRINT"AREA =";FNR(π*R↑2)
50 PRINT"VOLUME OF SPHERE =";FNR(4*π*R↑3
/3)
60 PRINT"AREA OF SPHERE =";FNR(4*π*R↑2)
```

Listing 10.4. How to define and use a function.

The new version has the rounding function defined in line 5. The rest of the program is the same as before except that the expressions used in the calculations are each made the argument of the function.

Now for a function of a function! If we have a function V to find

the volume of a sphere, and a function R to round off the answer, can we combine the two? Try Listing 10.5. Line 1Ø defines the rounding function R, while line 2Ø defines the function to calculate volume, given radius. In line 4Ø the volume is printed with nine significant figures. In line 5Ø the function R of function V rounds off the answer to the nearest whole number. So although it sounds highly abstract, a function of a function has its practical uses.

```
10 DEF FNR(N) = INT(N+.5)
20 DEF  FNV(R) = 4*π*R↑3/3
30 INPUT R
40 PRINT FNV(R)
50 PRINT FNR(FNV(R))
```

Listing 10.5. A function of a function!

Before leaving the rounding function here are the functions for rounding to other numbers of decimal places:
Rounding to 1 decimal place:

INT((N+.Ø5)*1Ø)/1Ø

Rounding to 2 decimal places:

INT((N+.ØØ5)*1ØØ)/1ØØ

Rounding to 3 decimal places:

INT((N+.ØØØ5)*1ØØØ)/1ØØØ

Defining one or more of these as functions can save a lot of memory space if rounding off is to be done at several points in a program.

Random numbers are often required in games programs and are helpful in giving variety to colour graphics displays with the minimum of programming. The Magic Carpet program (Listing 3.3) used five different random numbers to obtain its effects. If we are going to use random numbers at several stages in a program and if, as is likely, they are to be drawn from differing ranges, a 'random number function' could save a lot of program space. Try the short program in Listing 10.6.

```
10 DEF FNR(N) = INT(RND(1)*(N+1))
20 INPUT N
30 FOR J = 1 TO 100
40 PRINT FNR(N);
50 NEXT J
```

Listing 10.6. A handy random number function.

This program takes in a value N and then prints out 100 random numbers in the range Ø to N. At any stage in a program you can use this function to produce a random number in any range from zero to

N. Simply use a line similar to line 4Ø in the program above, substituting any number, variable or even a numerical expression for N. If you require that the range is from 1 to N, instead of Ø to N, use this definition of the function:

```
1Ø DEF FNR(N) = INT(RND(1)*N) + 1
```

Inputting 6 to this program simulates throwing a dice with sides numbered in the usual way, from 1 to 6.

To summarise

In this chapter, you have found out how to:

- use the 4 function keys in interactive programs
- use the function POS, to find where the next character is to be printed
- use the function SQR to find the square root of a number
- PRINT a circle on the screen
- calculate the co-ordinates of a body moving in a circular orbit
- convert degrees to radians and radians to degrees
- define your own functions and use them in programs
- round off numbers to any required degree of precision.

You have learned that:

- 'function' is an over-worked word with several distinct meanings
- π is both a symbol that can be PRINTed on the screen and a variable with a fixed value.

Chapter Eleven
Colour on the Move

Some of the earlier chapters have included programs which produce moving pictures on the screen. Now we are going to delve further into the techniques of the VIC's animated graphics. The special characters used in this chapter were all generated using the Newkeys or Multikeys utility programs described in Chapter Seven (Listings 7.3 and 7.4). It is well worth while to type in these programs and SAVE them on tape if you are thinking of writing your own programs with moving graphics.

Several different ways of producing moving pictures are described in this chapter, with an example of each. In all of them the key operation is to replace one picture with a picture which is slightly different. If this is done quickly enough, there is an illusion of motion. Sometimes the whole picture is moved, by PRINTing it in successive columns or rows. Then the whole object appears to move across the screen. Or we may move only a part of the object, by PRINTing a *slightly* different picture on the same area of the screen. The object appears to stay in the same place, but to move some of its parts. These two kinds of motion can be combined, so that the object not only moves its parts but also travels across the screen. There are examples of all these kinds of animation in this chapter.

Programming for moving graphics

This section applies to most, but not all, of the programs described in this chapter. These programs may be used in three different ways. All three ways use the same routines for putting the pictures on the screen and getting them to move.

If you have just used Newkeys or Multikeys (see (1) below), you can display the characters and write short programs with them. This applies until you press RUN/STOP with RESTORE or switch off.

A slightly different procedure is required when you want to use these characters again in your own programs. There are two ways of doing this (see (2) and (3) below) and which one you choose depends on how much memory you need for your program.

(1) After Newkeys or Multikeys

Begin by loading one of these utilities and running it. Each of the examples in this chapter has a drawing showing the characters required. Use these as a guide to defining the characters. The characters are allocated to the keys named beside each drawing. When you have finished defining all the characters for a program check that the eight codes for each character are the same as the codes written beside the drawings.

Newkeys and Multikeys both carry out a series of POKEs which make the VIC accept your new definitions instead of the characters normally associated with those keys. They also copy the upper-case and right-side characters into RAM, so that you may continue to use these. Provided that you do not press RUN/STOP with RESTORE (or switch off), the VIC retains this setting even after you have got rid of the Newkeys or Multikeys program by typing NEW. You can then enter and RUN the listings given in this chapter. You can also write variations on these listings or entirely new programs using the same characters.

However, if you save the program and then switch off, the program will not work when it is loaded the next time, for the VIC will then be using its built-in character generator and will PRINT Xs, Ys or Zs instead of the special characters. The solution to this problem is to use method 2 below.

(2) Independent programs

All that has to be done to make a listing independent of the utilities is to add a few extra lines to take care of the essential things that the utilities do (see Listing 11.1).

```
1 POKE 51,255:POKE 52,23:POKE 55,255:POK
E 56,23:POKE 36869,254
2 FOR J = 0 TO 1023
3 X = PEEK(32768 + J):POKE 6144 + J,X
4 NEXT J
5 FOR J = 0 TO 31
6 READ X
7 POKE 6328 + J,X
8 NEXT J
```

Listing 11.1. Putting user-defined graphics into your programs: how to begin the program.

These have been numbered so that they can be inserted *before* the listings, which all begin at line 1Ø. Line 1 sets the top of memory and string space, and then makes the computer take its characters from RAM (6144 onward) instead of from the character generator. Lines 2 to 4 transfer the codes for the capital letters and right-side graphics to RAM from the character generator ROM.

Lines 5 to 8 transfer the codes for each special character to RAM (from DATA statements) replacing the codes for capital letters. The DATA statements are typed in as extra lines at the end of each program. The figures needed for the DATA statements are those listed down the side of each character. For example, for the Talker program, the DATA statements are shown in Listing 11.2.

```
1000 DATA 126,255,187,153,255,231,231,12
6,66,126,0,0,0,0,0,0
1010 DATA 126,255,221,153,255,231,231,12
6,66,102,24,0,0,0,0,0
```

Listing 11.2. A typical DATA statement, to go at the end of the program.

The values given in the routines above allow you about 2 kilobytes of RAM for your program, and you have the facility of being able to use capital letters and right-side graphics in the normal way. The values in lines 5 and 7 are correct for a program such as Talker (Listing 11.7), which uses four special characters. These numbers need to be altered when the a program uses a different number of characters, as shown in Table 12.

The main snag about this method is that it takes 25 seconds to transfer all the keyboard characters. If you want to have only the user-defined characters and do not care about being able to have capital letters or right-side characters in your display, there is no point in wasting time by transferring all of these to RAM. You can always press RUN/STOP with RESTORE to get them back temporarily, for writing the program. To save transfer time, use the lines 2 to 4 in Listing 11.3 instead of those given above in Listing 11.1.

```
2 FOR J = 0 TO 7
3 POKE 6400 + J,0
4 NEXT J
```

Listing 11.3. Alternative lines for Listing 11.1 transfer only the 'space' to RAM.

The lines transfer only one of the normal characters, the space. This is essential for, without it, the screen has a mottled appearance instead of being a uniform colour.

(3) Independent programs with maximum memory

This is a variation on method (2) above, but puts the special characters further up in RAM. It leaves about 3 kilobytes for your programs. The keys produce capital letters, (except for those which have just been redefined as special characters) but there are no graphics characters. If you want to use any of the right side set in a display, it is easy to recover them. Begin each PRINT statement that is to contain right-side graphics with a 'reverse on' command, as shown in Listing 11.4. The 'reverse on' is typed by pressing CTRL

```
60 PRINT"■WE ♥ THE VIC!"
```

Listing 11.4. A typical PRINT statement for use with method (3).

with key 9. The heart, club or other symbol is typed in the normal way by pressing SHIFT with the appropriate key. You will not see any hearts, clubs or other recognisable symbols when you are typing in the program line; some unrecognisable assemblies of dots appear instead! Do not worry, for all is well. When the line is actually executed, the hearts and other symbols are PRINTed on the screen, just as you intended. They also appear properly in listings with the VIC 1515 printer. You can include capital letters with the symbols; these appear normally as you type and also on the screen when displayed.

```
1 POKE 51,255:POKE 52,27:POKE 55,255:POK
E 56,27:POKE 36869,255
2 FOR J = 0 TO 511
3 X = PEEK(32768 + J):POKE 7168 + J,X
4 NEXT J
5 FOR J = 0 TO 31
6 READ X
7 POKE 7352 + J,X
8 NEXT J
```

Listing 11.5. This version of Listing 11.1 uses less memory.

Listing 11.5 gives the lines to do at the beginning of all programs when you want maximum memory. These are almost the same as those used for method (2) above, except for certain values. As with method (2), the program takes several seconds to transfer the capital letters to RAM. Fortunately, it takes only half as long as method (2) for it is not having to transfer right side symbols. If you want to eliminate this stage, transfer the space only, by substituting the lines in the Listing 11.5.

```
2 FOR J = 0 TO 7
3 POKE 7424 + J,0
4 NEXT J
```

Listing 11.6. Use these lines to begin your program, in method (3).

The values in lines 5 and 7 are for use when there are four user-defined characters. For other numbers of characters use the values given in Table 13. As with method (2), the details of the characters are entered as DATA lines at the end of the program. These lines are exactly the same as used in that method.

It may seem that programming for colour graphics is a complicated matter, but this is only because several alternative methods are being described. It is best to stick to one method until you have become used to it. If you are a beginner, use method (1). You need only load Newkeys or Multikeys (Listings 7.3 and 7.4) and then define the characters. All you have to do after that is to type in the listings given in the rest of this chapter. You will enjoy watching the results. Later, when you are more experienced and want to write your own programs, try methods (2) or (3), perhaps using method (2) at first for it is more convenient. When your programs are tending to be longer, you might switch to method (3).

Moving parts

The Talker program (Listing 11.7) shows a skull which opens and closes its jaw repeatedly, at the same time rolling its eyes from side to side. The picture consists of two pairs of graphics characters (Fig. 11.1). These are referred to by the letter of the key to which they are

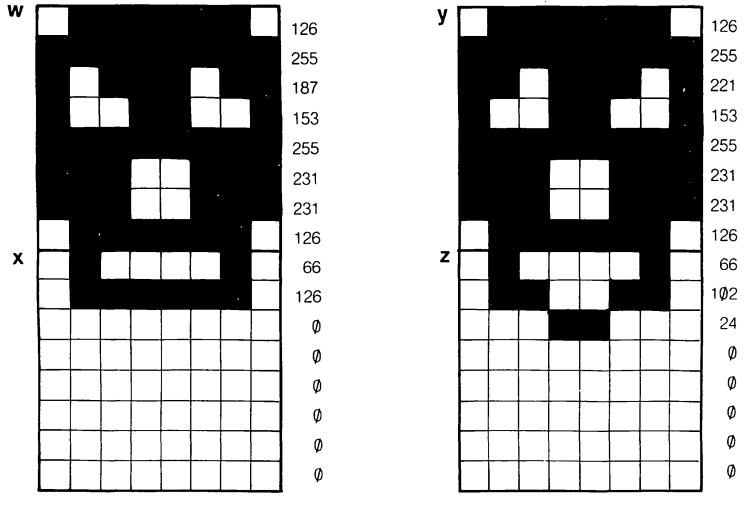


Fig. 11.1. Characters for the Talker program.

allocated. Two of these characters are PRINTed one above the other at the centre of the screen. When W is PRINTed above X we see a skull with closed jaw and eyes looking toward the right of screen. When Y is printed above Z, we see the skull with open jaws and eyes looking toward the left. If W-above-X and then Y-above-Z are PRINTed alternately in rapid succession, the skull appears to talk and roll its eyes.

```
10 PRINT"□■":POKE36879,8
20 PRINT"■■■■■■"
30 PRINTTAB(10)"W"
40 PRINTTAB(10)"X"
50 FOR J = 1 TO 200:NEXT J
60 PRINT"■"
65 PRINT"■■■■■■"
70 PRINT"■■■■■■"
80 PRINTTAB(10)"Y"
90 PRINTTAB(10)"Z"
100 FOR J = 1 TO 200:NEXT J
110 PRINT"■":GOTO 20
```

Listing 11.7. The Talker.

Line 1ϕ sets the PRINTing colour to white and both screen and border to black, so obtaining the required spooky effect. Lines 2ϕ and 7ϕ make the cursor move five rows down the screen before PRINTing the characters. To type this line, first key in PRINT", then press the up-down cursor key (unshifted) five times. Then type ". Lines 6ϕ and 11ϕ send the cursor back home without clearing the screen (CLR/HOME unshifted). You can see in lines, 3ϕ, 4ϕ, 8ϕ and 9ϕ that the characters appear as W, X, Y and Z in the listing. But on the screen they appear as a talking skull.

Motion and locomotion

Figure 11.2 shows the designs for characters which produce the illusion of a person walking across the screen. There is one set of characters, named U, and V, and a second set named W to Z. U and V represent the person having just taken a stride forward. It would be possible simply to PRINT U and V in successive columns across the screen, but this would not work. It worked well with Puffer (Listing 7.1) because, at the scale the locomotive was drawn, one would not expect to see the wheels going round and its pistons reciprocating. In fact its wheels had no spokes and its pistons were not drawn, so there was nothing to spoil the illusion. The person has legs, which are clearly visible and which we expect to see performing a walking action. If we simply plot U and V in successive columns, the person appears to skid across the screen as if on ice.

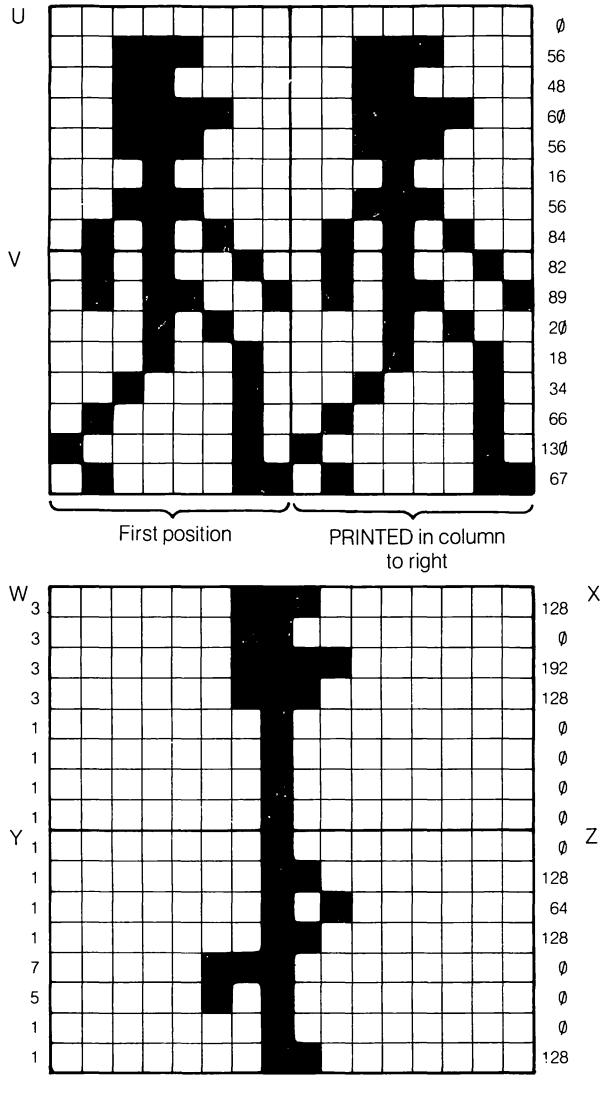


Fig. 11.2. Characters for the Walker program.

W to Z provide the link between two successive PRINTings of U and V. The person is stepping from one column of the screen to another, and appears partly in one and partly in the next. This is why four characters are needed. Comparing the two sets of characters, in the second set we see that the forward foot has slipped forward slightly, as if being finally placed on the ground. The rear foot is being brought forward and the leg is bent at the knees as this happens. The arms are not shown because they are hidden by the body as one swings forward and the other swings backward. The

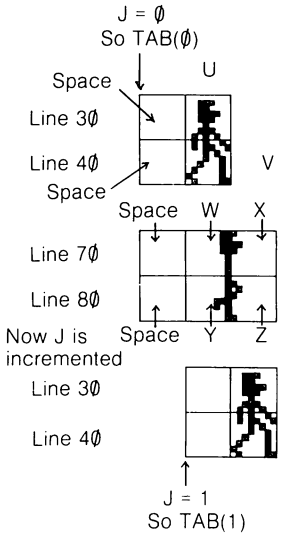


Fig. 11.3. The way the figure is made to walk across the screen.

head rises because the person is vertical at this stage of locomotion.

The order of PRINTing the characters is shown in Fig. 11.3. They are PRINTed as strings. These include spaces to block out previously PRINTed characters. The spaces PRINTed before W/X and Y/Z are not strictly necessary as W and Y overprint U and V. However, if we leave out these spaces, the TABs in lines 70 and 80 have to be amended to TAB(J + 1), so it is easier just to include the extra space. The program is shown in Listing 11.8.

```
10 PRINT"[graphics]"
20 FOR J = 0 TO 18
30 PRINTTAB(J)" U"
40 PRINTTAB(J)" V"
50 PRINT"[graphics]"
60 FOR K = 1 TO 100:NEXT K
70 PRINTTAB(J)" WX"
80 PRINTTAB(J)" YZ"
90 PRINT"[graphics]"
100 FOR K = 1 TO 100:NEXT K
110 NEXT J
```

Listing 11.8. The Walker.

PRINT or POKE?

PRINTing the characters on the screen has the advantage that it is simpler to write the program and it is generally easier for other people to read and understand the program. One disadvantage is that PRINTing may interfere with other parts of the display.

POKEing puts the characters where they are required without affecting any other screen locations. Also it is often easier to move characters around the screen using loops containing POKE statements and variables than it is to perform similar operations with PRINT statements. Listing 11.9 shows a version of the Walker program which uses POKE instead of PRINT.

```
10 PRINT"□":D = 30720
20 FOR J = 7724 TO 7742
30 POKE J+D,1:POKE J+D+1,2:POKE J+1,21
40 POKE J+D+22,1:POKE J+D+23,2:POKE J+23
,22
60 FOR K = 1 TO 100:NEXT K
70 POKE J+D+2,2:POKE J+1,23:POKE J+2,24
80 POKE J+D+24,2:POKE J+23,25:POKE J+24,
26
100 FOR K = 1 TO 100:NEXT K
110 NEXT J
```

Listing 11.9. This version of the Walker uses POKE instead of PRINT.

If you compare this with Listing 11.8, it is easy to spot the similarities. W is given the value of the first screen location into which the top half of the person is to be POKEd. The variable D is the difference between addresses in character code RAM and colour code RAM (see Fig. 3.3). Any command involving D is one that determines the colour in which a character is to be displayed.

The commands in line 3Ø do the following:

POKE J + D, 1	Display in white - since the background is white, this clears away a previously visible character. It is the equivalent of PRINTing a space.
POKE J + D + 1, 2	Display in red - the program shows a red person.
POKE J + 1, 1Ø	Make the top half of the person appear in screen location J. The top half is defined on key 'U', so to find the number to POKE, look at Table 6 in Appendix A. There you find that the screen code for 'U' is '21'.

The commands in line 4Ø are similar, but refer to the bottom half of the person. The addresses POKEd are all 22 more than those in line 3Ø to make the bottom half appear directly below the top half. The screen code for 'V' is '22'. At lines 7Ø and 8Ø characters W to Z are POKEd into their correct positions with respect to U and V, while the original U and V are made invisible by causing them to be displayed in white.

Collisions

If people and objects are to be moved around the screen in a realistic way, they must behave as *real* people and objects do. In particular, when they come into contact with some other person and object, they must stop, or bounce off them or in some other way behave as if they have collided. This is essential in games programs in which we may want to make a ball bounce off a wall, or to make a bat hit a ball. We may also want to know when a missile has hit its target. In the Cupid game (Listing 11.10), the missiles are Cupid's arrows and the targets, naturally enough, are hearts.

```
10 PRINT"■■":POKE 36879,205:R = 8164
20 FOR J = 1 TO 20
30 PRINTSPC(INT(RND(1)*20));"♥"
40 NEXT
50 GET A$
60 IF A$ <> CHR$(133) THEN GOTO 50
70 FOR J = 0 TO 65
80 GET A$:IF A$ = CHR$(136) THEN R = R -
 22
90 IF PEEK(R+J)=83 THEN GOSUB 1000
100 POKE R + J,30:POKE R+30720+J,5
110 FOR K = 1 TO 500: NEXT K
120 NEXT J
130 PRINT"■■";TAB(2)"YOU HIT";N;"HEARTS!
"
140 GOTO 140
1000 N = N + 1:POKE 36878,15:T = 150
1010 FOR L = 1 TO 50
1020 T = T - 6
1030 POKE 36876,T
1040 FOR M = 1 TO 10:NEXT M
1050 T = T + 8
1060 POKE 36876,T
1070 FOR M = 1 TO 10: NEXT M
1080 NEXT L
1090 POKE 36876,0:POKE 36878,0
1100 RETURN
```

Listing 11.10. Cupid, a game which shows you how to detect collisions.

Since this program uses VIC graphics characters and not user-defined ones, the methods described in previous sections do not apply. Just type it in and RUN it.

Before discussing collisions between arrows and hearts there is a new function to look at in line 3Ø. SPC(N) is used in PRINT statements to cause N spaces to be PRINTed. In line 3Ø the argument of SPC is a random number between zero and 19. The effect of lines 2Ø to 4Ø is therefore to scatter the screen with 20 hearts. The pattern of hearts on the screen is different every time the program is RUN. The GET A$ at line 5Ø gives the user time to inspect the screen and to plan tactics. When the player presses function key F1 the game begins.

The loop from line 7ϕ to 12ϕ PRINTs a series of green up-arrows on the screen. PRINTing is always at location R and R is set to 8164 to begin with (line 1ϕ) so that the first arrow comes at the bottom left-hand corner. R is incremented each time through the loop, so that the arrows are PRINTed in successive columns across the screen and, with wrap-around, the screen is traversed three times. When the player presses key F7 (line 8ϕ), R is reduced by 22, causing the arrow to be PRINTed on the next line above. Thus the row of arrows is horizontal (F7 not pressed) or slopes up to the right (F7 pressed).

Collisions are detected in line 9ϕ by PEEKing the location (at R + J) in which the next arrow is going to be PRINTed. If PEEK(R + J) is found to be '83', the screen code for a 'heart', a collision is unavoidable, for in the next program line that same location will be POKEd with the screen code for an arrow. However, before striking the target, the program jumps to a subroutine to keep score of the number of hearts hit (N) and to produce a sound effect.

When the main loop has been repeated 66 times and the row of hearts stretches three times across the screen, the game ends with a display of the final score (line 13ϕ). Press RUN/STOP with RESTORE to play again. It takes practice to work out the best strategy for each arrangement of hearts. When you become more proficient, make the game quicker by reducing the length of the delay loops in line 11ϕ.

Moving colours

The usual way of making an object move around the screen is to PRINT it or POKE it to one location on the screen after another. PRINTing may affect other images already on the screen, while POKEing is a little complicated in that both character code RAM and colour code ROM must be POKEd each time. Here is a way of simplifying the procedure and speeding up the operation. First of all, the character code is POKEd to *all* locations to which the object is to be moved. Assuming that the screen is white, nothing appears on the screen at that stage. When the object is due to appear at a given location the *colour code RAM* of that location is POKEd with the required (non-white) colour. The object appears immediately! To make it move to the next location its current colour code location is POKEd with 1 to display it in white (that is to say, to make it disappear), while the next colour code location is POKEd with its

colour. Disappearing from one location and reappearing at the next location gives the illusion of motion.

The program in Listing 11.11 demonstrates the technique. This is

```
10 PRINT"□"
20 FOR J = 7724 TO 7745
30 POKE J,81
40 NEXT J
50 FOR J = 38444 TO 38465
60 POKE J-1,1:POKE J,5
70 FOR K = 1 TO 200: NEXT K
80 NEXT J
```

Listing 11.11. Creating the illusion of motion by simply changing colours.

another program which is typed in and RUN. Lines 2ϕ to 3ϕ POKE a 'disc' into all the character code locations of the third row of the screen. Nothing appears on the screen at this stage but everything is now set up for action. At lines 5ϕ to 7ϕ the locations of colour code RAM for the third screen row are POKEd with '5' one after another to cause the 'disc' to appear in green. At each stage, the (J−1)th location is POKEd with '1' to make the preceding disc disappear at the same time as each new one appears. The effect is that of a ball moving across the screen. This program allows the ball to travel along one row simply by POKEing one location each time. You can add lines to the program to send the 'ball' back and forth along the row as many times as you like without having to POKE the ball shape again. Simply 'move the colours'.

As a variation on this program alter lines 2ϕ and 3ϕ as shown in Listing 11.12. This places a 'disc' and a horizontal mark in alternate

```
20 FOR J = 7724 TO 7745 STEP 2
30 POKE J,81:POKE J+1,67
```

Listing 11.12. The Jellyfish.

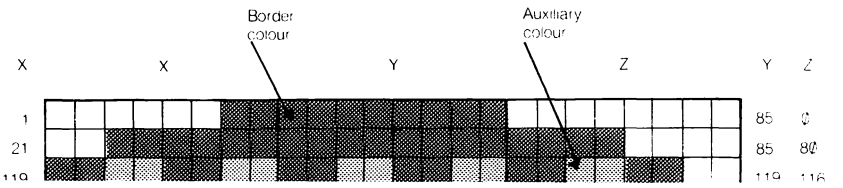
locations along the third screen row. Now, when the program is RUN, the disc appears to squash itself flat as it moves, pulsating like a jellyfish. By suitable choice of characters, either those provided by the VIC or those you design for yourself, you can create many interesting effects by this technique.

Multicolour motion

This technique takes the idea of the previous one and exploits multicolour graphics to improve upon the effect. Figure 11.4 shows the design for a flying saucer such as might turn up on your screen from outer space at any moment. It consists of three multicolour characters, defined by using the Multikeys utility program (Listing 7.4).

character X. When the value of R is changed later in the program, the characters will still be printed in their respective positions. Thus we can say exactly where the UFO is to appear on the screen simply by altering the value of R. The values given by 24 + J are the screen codes for each of three characters in turn.

The effect of lines 2ϕ and 3ϕ is therefore to display the UFO at the



bottom of the screen, with rockets off and lights on. There is a short delay (line 4Ø) and then the lights are turned off by making the auxiliary colour change to black, that is to say, by POKEing Ø to 36878. Now there is a chance for the user to interact with the program by pressing the function key F7. If this is *not* pressed, the program drops through to line 8Ø, where there is a further delay, holding the lights off. Then line 9Ø puts the lights back on again and there is a jump back to line 4Ø to repeat the sequence. The lights flash on and off steadily.

If the user presses key F7, the commands in line 7Ø come into effect. Giving N the value 6 makes the foreground colour blue, the same as the background. The subroutine at 1ØØØ POKEs this to the locations of the UFO in colour code RAM. Note that this time we are using the colour code for blue (6) *without* adding 8 to it. The effect is to switch off the multicolour mode for these locations. As a result, the characters are treated as ordinary two-colour characters. They are displayed in the blue foreground colour on a blue background – they become invisible! Then R is decremented, by 22, to take the UFO one row up the screen, and the subroutine at 2ØØØ places the characters in this row. N is made equal to 1Ø (2 for colour red + 8 for multicolour mode) and the micro is sent back to line 2Ø to display the UFO in its new position. Now, with red as the foreground colour, its rocket engines are burning. Each time F7 is pressed the UFO rises to the row above and then hovers, with its rockets aglow and its lamps flashing in a menacing manner against the evening sky.

(*Note* for those who cannot resist adding sound effects: The values for the only possible tune to accompany the flight of the UFO are 214, 218, 2Ø9, 162 and 193, played in that order by the bass tone generator. If you want to add to the illusion, change the colour of the UFO's lights in sympathy with each note of the tune. The colours are (in order): yellow, pink, purple, blue and red.)

In this example, the object is moved across the screen by changing the value of its reference point R. At the same time we are able to alter it appearance by changing the foreground colour and auxiliary colour. Changing the auxiliary colour allows you to change the appearance of several different objects located at different parts of the screen by using a single POKE command.

A single POKE to change the auxiliary colour does the trick. In this program, we are simply turning lamps and rockets on or off, but the idea can be extended to movement of parts of the object as the example in the next section (Dancing Puppet) will show.

The program in Listing 11.13 is typed in after defining the three characters with Multikeys and then typing NEW. The characters were assigned to keys 'X', 'Y' and 'Z'.

```
10 PRINT"□":POKE 36879,104:R = 8173:N =
14
20 GOSUB 1000
30 POKE 36878,112:GOSUB 2000
40 FOR K = 1 TO 100:NEXT K
50 POKE 36878,0
60 GET A$
70 IF A$ = CHR$(136) THEN N = 6:GOSUB 10
00:R=R-22:GOSUB 2000:N = 10:GOTO 20
80 FOR K = 1 TO 100:NEXT K
90 POKE 36878,112
100 GOTO 40
1000 FOR J = 0 TO 2
1010 POKE R + 30720 + J,N
1020 NEXT J
1030 RETURN
2000 FOR J = 0 TO 2
2010 POKE R + J,24 + J
2020 NEXT J
2030 RETURN
```

Listing 11.13. UFO program using multicolour graphics to put the UFO's rockets under your command.

Line 1Ø clears the screen, sets the screen colour to blue and the border to black. Then it sets N the foreground colour to blue (rockets off). N is the sum of 6 (blue) plus 8 (= multicolour mode). The variable R is the 'reference' location of the UFO in character code RAM. Here it is the ninth column on the bottom row of the screen. Line 2Ø sends the micro to the subroutine at lines 1ØØØ to 1Ø3Ø, which POKEs N into the colour code RAM, so that the foreground colour will be blue, the same as the background.

Line 3Ø sets the auxiliary colour to yellow (code = 7) by POKEing 112 (= 7 × 16) to address 36878. When the UFO is displayed, its yellow lights will be on. The micro then goes to the subroutine at line 2ØØØ. This subroutine has a rather different purpose from the one at line 1ØØØ. It POKEs the three characters, X to Z into their correct positions in character code RAM, using R as the address of character X. When the value of R is changed later in the program, the characters will still be printed in their respective positions. Thus we can say exactly where the UFO is to appear on the screen simply by altering the value of R. The values given by 24 + J are the screen codes for each of three characters in turn.

The effect of lines 2Ø and 3Ø is therefore to display the UFO at the

bottom of the screen, with rockets off and lights on. There is a short delay (line 4Ø) and then the lights are turned off by making the auxiliary colour change to black, that is to say, by POKEing Ø to 36878. Now there is a chance for the user to interact with the program by pressing the function key F7. If this is *not* pressed, the program drops through to line 8Ø, where there is a further delay, holding the lights off. Then line 9Ø puts the lights back on again and there is a jump back to line 4Ø to repeat the sequence. The lights flash on and off steadily.

If the user presses key F7, the commands in line 7Ø come into effect. Giving N the value 6 makes the foreground colour blue, the same as the background. The subroutine at 1ØØØ POKEs this to the locations of the UFO in colour code RAM. Note that this time we are using the colour code for blue (6) *without* adding 8 to it. The effect is to switch off the multicolour mode for these locations. As a result, the characters are treated as ordinary two-colour characters. They are displayed in the blue foreground colour on a blue background – they become invisible! Then R is decremented, by 22, to take the UFO one row up the screen, and the subroutine at 2ØØØ places the characters in this row. N is made equal to 1Ø (2 for colour red + 8 for multicolour mode) and the micro is sent back to line 2Ø to display the UFO in its new position. Now, with red as the foreground colour, its rocket engines are burning. Each time F7 is pressed the UFO rises to the row above and then hovers, with its rockets aglow and its lamps flashing in a menacing manner against the evening sky.

(*Note* for those who cannot resist adding sound effects: The values for the only possible tune to accompany the flight of the UFO are 214, 218, 2Ø9, 162 and 193, played in that order by the bass tone generator. If you want to add to the illusion, change the colour of the UFO's lights in sympathy with each note of the tune. The colours are (in order): yellow, pink, purple, blue and red.)

In this example, the object is moved across the screen by changing the value of its reference point R. At the same time we are able to alter it appearance by changing the foreground colour and auxiliary colour. Changing the auxiliary colour allows you to change the appearance of several different objects located at different parts of the screen by using a single POKE command.

Dancing Puppet

This is an example of how we use the foreground and auxiliary colours to produce the effect of moving the parts of an object. The puppet (Fig. 11.5) alternates between two positions (we use screen

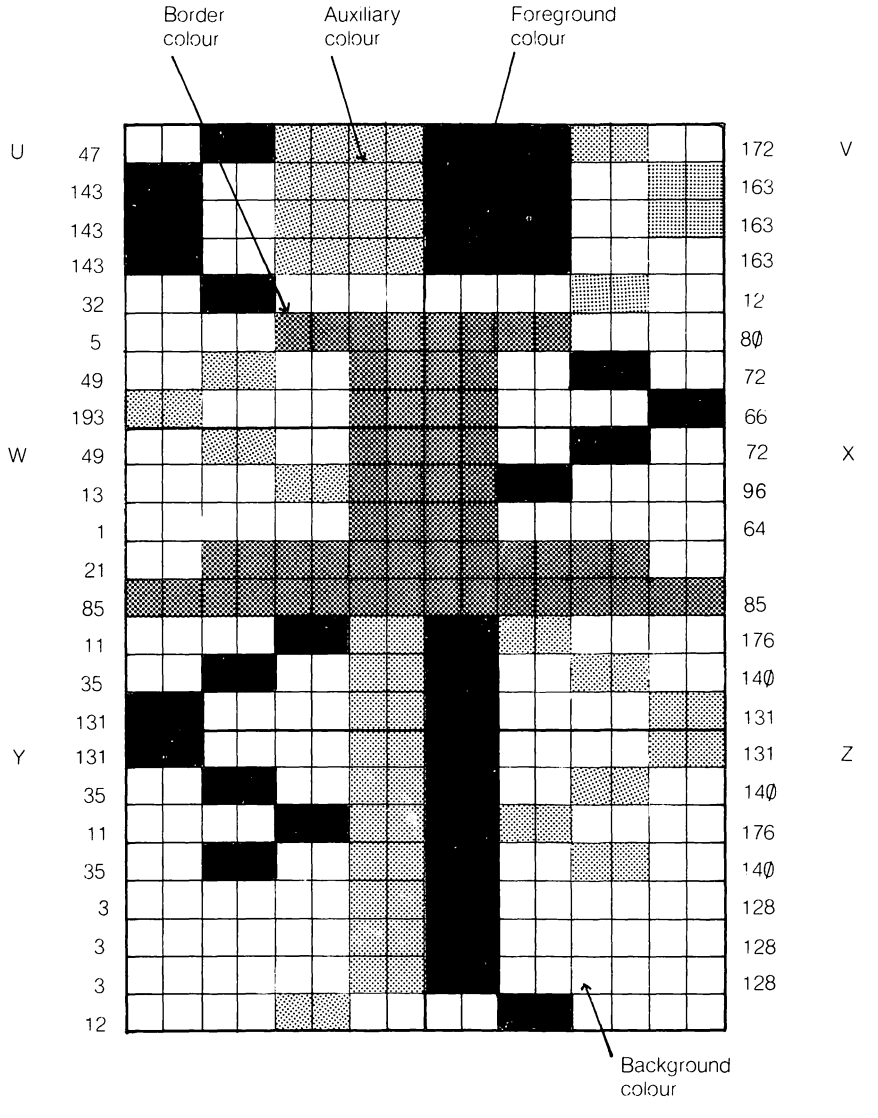


Fig. 11.5. Characters for the puppet.

left and screen right in describing them):
(1) Head to right, right arm down, left arm up, left leg raised.

(2) Head to left, left arm down, right arm up, right leg raised.

The four colours are used like this:

Background colour: as background (cyan).

Border colour: for parts of the puppet which do not move, such as the body and skirt (red).

Foreground colour: for the moving parts of the body in position 1 (red when the puppet is in position 1, cyan when in position 2).

Auxiliary colour: for the moving parts of the body in position 2 (red when the puppet is in position 2, cyan when in position 1).

The puppet is made to dance by switching from:

foreground = red *and* auxiliary = cyan
to auxiliary = cyan *and* foreground = red

This causes the two images to appear alternately, giving the illusion of motion. The program is shown in Listing 11.14.

```
10 PRINT"□":POKE 36879,58
20 R = 7800
30 GOSUB 1000
60 POKE 36878,48:N = 10:GOSUB 2000
70 FOR K = 1 TO 200: NEXT K
80 POKE 36878,32:N = 11:GOSUB 2000
90 FOR K = 1 TO 200:NEXT K
100 GOTO 60
1000 X = 21
1010 FOR J = 0 TO 44 STEP 22
1020 POKE R + J,X:POKE R + 1 + J, X + 1
1030 X = X + 2
1040 NEXT J
1050 RETURN
2000 FOR J = 0 TO 44 STEP 22
2010 POKE R + 30720 + J,N:POKE R + 30721
 + J, N
2020 NEXT J
2030 RETURN
```

Listing 11.14. The puppet, another program making use of multicolour graphics for animation.

Like the previous program, it has two subroutines. One (at 1000) POKEs the characters into character code RAM at a position determined by the reference location R. The other (at 2000) POKEs colour code RAM to control the foreground colour (N) and switch on multicolour mode.

Line 20 sets the reference location, and line 30 sends the micro to subroutine 1000 to POKE the characters into RAM. A subroutine is not really needed in this example since this operation is performed only once. But if you want to modify the program to move the puppet around the screen while it dances, it is useful to be able to change R and have this standard subroutine for making the puppet

appear in its new location. In this program the puppet dances in a fixed spot, and once the characters have been placed in RAM, all that is necessary to make the dance commence is to change the foreground and auxiliary colours. Lines 6Ø and 8Ø do this, with delays at lines 7Ø and 9Ø between each dance position.

Delays are used in these programs to give the viewer time in which to see each image. In a games program or a program of some other type using moving pictures, these delays would normally be put to good use by the micro. During the times while the object is stationary the micro has plenty of time to attend to the other routines of the game: animating other objects, moving them around the screen, reading the keyboard for a key-press, adding up the score, and many other jobs. It returns briefly at regular intervals to alter the picture to keep the object moving.

The monsters

The limitation of the technique above is that the object must be all of one colour. This final example (Fig. 11.6) shows how you can have

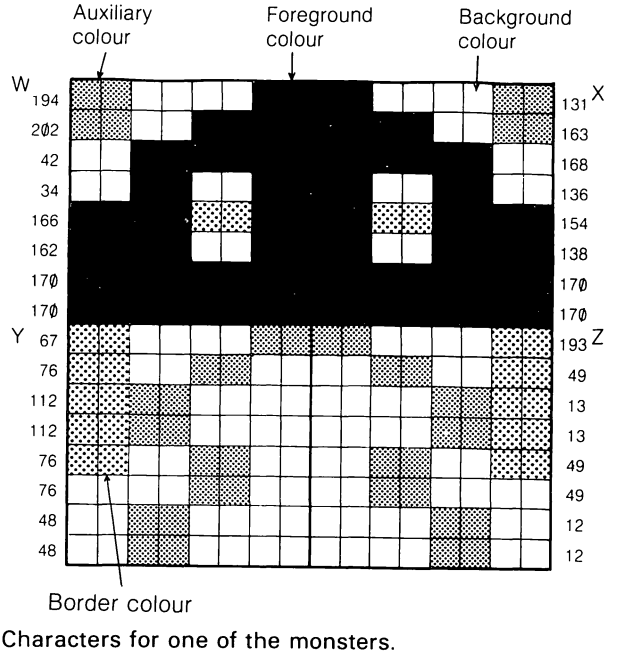


Fig. 11.6. Characters for one of the monsters.

two colours, yet still get a reasonable degree of motion. Here the motion is 'there' and 'not there' or 'appear' and 'disappear' rather than a true movement of parts from one position to another. The monster has two trailing tentacles below and two stubby feelers above, which it protrudes and retracts alternately. These are in the auxiliary colour. By alternately making this colour first red and then pink (the same colour as the background), the tentacles and feelers are made to appear and disappear.

Animating the figures is simply a matter of POKEing two different values alternately to address 36878 (see Listing 11.15). The characters are POKEd once and for all in lines 3Ø to 5Ø. This is

```
10 PRINT"□":POKE36879,172
20 R = 7705
30 POKE R,23:POKE R+1,24:POKE R+22,25:PO
KE R+23,26
40 R = R + 30720
50 POKE R,8:POKE R+1,8:POKE R+22,8:POKE
R+23,8
70 POKE 36878,32
80 FOR K = 1 TO 200: NEXT K
90 POKE 36878,160
100 FOR K = 1 TO 200: NEXT K
110 GOTO 70
```

Listing 11.15. POKE the Monster and it appears to move.

animation reduced to its simplest and quickest, and all of it is done in lines 7Ø and 9Ø. Lines 8Ø and 1ØØ provide the delays. With so little to do in the way of animating, the micro has plenty of time to spare for other tasks connected with the main program of which the monsters form but a part.

The earlier part of the program POKEs the characters to character code RAM and colour code RAM once and for all. R is the reference location. The body of the monster is in foreground colour (black, so we POKE Ø + 8 = 8, in line 5Ø). Its eyes and two shorter tentacles are in the border colour, purple, You can make the monster change its body colour by repeating line 5Ø and POKEing a different value.

As a final flourish, why not have a horde of monsters spread across the screen? Alter line 2Ø as shown in Listing 11.16, and add

```
20 FOR R = 7705 TO 8025 STEP 80
```

Listing 11.16. This line and the one on page 157 ...

```
60 NEXT R
```

Listing 11.17. ... give you the Multi-Monster horde!

line 6ø (as Listing 11.17).
Now you have half a dozen monsters, animated!

To summarise

In this chapter, you have found out how to:

- animate a picture by displaying two different pictures alternately
- animate a picture and move it across the screen at the same time
- detect collisions
- move an object across the screen by changing its colour
- animate pictures by using multicolour mode.

Chapter Twelve

The Organised Programmer

This chapter covers an assortment of topics aimed at helping you to get the best out of your VIC 20.

REM

There must certainly be plenty of readers who are wondering why the mention of this statement has been left so late. All these pages of programs and not a single REM! To make up for the deficiency, an example of its use is shown in Listing 12.1.

```
10 REM THIS PROGRAM IS WRITTEN FOR
20 REM THE VIC20 MICROCOMPUTER
30 LET X = 99
40 PRINT X
```

Listing 12.1. Using REMs.

There is more to this program that you might think. Let us consider the REM first. Considering REMs is something that micros never do. REM is short for 'REMark' and the remark is directed to the person reading the listing, not to the computer. Whenever the computer finds the word 'REM' on a program line, it ignores the rest of the line and goes on to the next. In the program above, the VIC does not start taking interest in the program until it gets to line 3Ø.

People have widely differing opinions about the use of REM. Some are strong believers in its usefulness. They begin their programs with REM lines which give the title of the program, its purpose, often their own name as author, and frequently detailed instructions on how to use it. Then they may have REMs scattered throughout the program, explaining briefly what is happening at each stage. Every subroutine will begin with a REM to state what it does. There is no doubt that REMs can help other people to read the

program with understanding. You may also find that REMs help you to remember how your own programs work. Although you may be very clear about this when you have just finished writing a program, it is surprising how quickly one forgets the details. Even a week later, the program may appear almost incomprehensible to you, its creator!

One reason that there are no REMs in the programs in this book is that there is the text which explains the programs far better than the somewhat terse comments of a REM can ever do. Another reason is that since these are mostly short examples, to be typed in quickly and tried out just once, there is no point in giving you all that extra typing. Another reason for avoiding REMs, or at least making them as short as possible, is that they use a lot of memory. Whereas BASIC keywords are stored as coded tokens, each word taking one byte, every letter, figure, space or symbol in a REM takes a whole byte. If you have only about 3.5 kilobytes to use, you do not want to waste a high proportion of it on information which the computer always ignores. Another point about REMs is that, even if the whole of a program line is a REM and the computer does not do anything about it, it still has to look along the line as far as the word REM. This takes time. Too many REMs slow down the speed at which the program runs.

It is clear from the above that programmers may have divided opinions over the matter of REMs. Some are strongly for, others are strongly against. Both sides of the discussion have been put to you, so that you can make up your own mind on this subject. Perhaps the middle road is the wisest choice. A REM on the first line, as a title, is always helpful in identifying a program. It also helps to remind you of its *exact* title, giving you the correct name for loading it. REMs are also helpful to separate the main sections of programs, particularly where there is a long jump from one part of a program to another, or to a subroutine. However, it is a good rule never to jump to a line beginning with a REM, or to begin a subroutine with a REM. The REM is best put on the line *before* the part of the program or subroutine begins (see Listing 12.2).

```
999 REM POKES CHARACTER CODES TO RAM
1000 FOR J = 0 TO 3
1010 READ X
```

Listing 12.2. Where to put REMs in your programs.

The reason for this is that if you or someone else is typing in the program, and wants to leave out the REMs to save space or effort,

it is a simple matter to leave out a line such as line 999. If the REM was on line 1ØØØ and you left out this line, there would be no line for the micro to jump to when the subroutine was called. This would result in an error message. If you leave out the line, you have to go through the program renumbering all the GOTOs and GOSUBs. The recommended method eliminates these possible sources of error.

Whether you belong to the REM school of thought, or the non-REM school, it is essential that all the programs you write should be properly documented. If you prefer to leave out REMs, then keep full notes on paper. Write down the main stages of the program, what they do and how they work. If you prefer, make a flow chart (see Figs 5.6, 6.1 and 8.1) to show the order in which things happen. It is sometimes stated that you should always draw out a flow chart *before* you begin to write the program. Many people find this a helpful thing to do, and there are certainly programs or parts of programs for which it is beneficial to begin by drawing a flow chart. But, in practice, many programmers prefer to plan their programs by writing out the main stages as short sentences, perhaps with arrows leading from one to another. The flow chart is drawn later, when the program has been *finished* and tested, as a final record of its main features. 'Flow charts before' or 'flow charts after' – another choice for you to make. Think about it, work out which system is best for you, and keep to it.

When making notes about your program, go through it from beginning to end, writing down the name of every variable and what it does. It is helpful to draw a diagram of every array (see Fig. 9.2) to show what each row and column is supposed to contain.

LET X = 99

This short line, taken from the program in Listing 12.1, is the next subject for discussion. LET is yet another BASIC word which has been left until this last chapter. The reason is that there is no need to use it. If you type it in, the VIC understands it and acts on it. If you leave it out and just type:

```
X = 99
```

the VIC assumes that the LET is really there and obligingly 'lets' X become equal to 99. Most micros, though not all, do not require you

to use the word LET. Perhaps this is a pity for it is really a very important word. Look at this line:

```
10 N = N + 3
```

This kind of line has been used often in this book. We say that it increments N by 3. But look again – this is nonsense! How can N be equal to N + 3? The rules of mathematics make such an equality impossible. If we type the full statement, as the VIC understands it, the true meaning becomes clearer:

```
10 LET N = N + 3
```

The meaning is: 'LET the variable N take the value which is obtained by adding 3 to its present value'. The 'equals' in this expression does not mean 'equals' in the way that we generally use it in mathematics. It is being used to *assign* a new value to N. To obtain the sense of its meaning, replace the equals sign by the word 'become': 'LET N become the sum of N and 3'.

This confusion over the use of the equals sign would be easier to get straight if the sign were used *only* in this sense in BASIC, but it is not. In an expression such as 'IF X = 5 THEN...' it has its normal mathematical use in comparing one quantity with another – it is being used as a relational operator. It is important to get these two distinct uses of the equals sign clear in your mind.

Most programmers leaves out LET, if their micro allows it. There are other words which can be left out in given circumstances. Leaving them out makes the program less easy to read, especially for a beginner, but one soon gets used to this. For example, it is generally unnecessary to use the word 'GOTO' in an 'IF...THEN GOTO line-number' statement. Instead of

```
20 IF X = 5 THEN GOTO 300
```

it is in order to write:

```
20 IF X = 5 THEN 300
```

The micro interprets the number following the 'THEN' as a line number.

```
10 FOR J = 1 TO 5
20 FOR K = 3 TO 6
30 FOR L = 10 TO 3 STEP -2
40 PRINT J*K*L
50 NEXT
60 NEXT
70 NEXT
```

Listing 12.3. You may leave out the variable names after NEXT.

Another omission which does not upset the computer is the variable following the word NEXT. For example see Listing 12.3. The micro does not worry about the missing variables in lines 5Ø to 7Ø. This program can be made more compact by putting all the NEXTs on one line as in Listing 12.4.

```
50 NEXT : NEXT : NEXT
```

Listing 12.4. All the NEXTs may be placed on one multistatement program line.

Although it is easy enough for the reader to keep track of the NEXTs in a short and simple program such as the above, it becomes more difficult with a longer program which has many loops. The computer automatically counts how many FORs and how many NEXTs it has met and always informs you by an error message if you have put too many of one or the other. But a person checking through the program can soon become confused. So, unless you are anxious to save every last byte of memory, it is preferable retain the variables after the NEXTs.

Saving memory

Although the newer micros have memories of a size that a few years ago would have been prohibitively expensive, there is a variant of Parkinson's Law which states that 'A program expands to take up all the RAM available to hold it'. Just a few extra routines and refinements to that already 'perfect' program and yet another 2 kilobytes is swallowed up! So it is worth while considering a few general rules for saving memory space:

(1) Leave out all REMs except those which are absolutely essential.
(2) Leave out spaces between words in program lines. In the listings in this book, spaces have been used liberally in order to make it easier for you to read the programs, but a line such as

```
2ØØPOKE36879,27:B(8,N)=5632+8*ASC(K$(N)):N=N+1
```

is easily understood by the VIC, even if it looks somewhat indigestible to you. Leaving out the spaces helps you get more statements on a line (see (3)).
(3) Use multistatement lines (for example, the one just given in Listing 12.4). The statements are separated from each other by colons(:). The VIC allows you to have up to 84 characters on a line,

including spaces, punctuation and symbols, but not including the space which automatically appears (on LISTing) between the line number and the first character typed in.

(4) Use the 'zero' rows and columns in arrays. The top row of array A() of Fig. 9.2 is a waste of RAM, for the zeros there are being stored in memory, even though they are never used by the program. It would have been more economic to have used this space, though it might have made the working out of the loop which uses this array a little more difficult to understand. When you become familiar with arrays, it becomes easier to make use of the zero rows and columns without confusion. Using the 'zero' rows and columns is especially important if you have very large arrays.

(5) Use subroutines, instead of repeating the same routines wherever they are needed through the program.

(6) Use integer variables (Chapter Four). These take up much less storage space than floating point variables. This could make big savings if you have large arrays in your program. For example, if you extended the FXG utility (Listing 9.1) to cover many more phases, it would be worth making array A() into array A%(), for all the numbers required for controlling the sound generators are integers.

Faster BASIC programs

There are several ways of making the program run as quickly as possible.

(1) Leave out all REMs that are not absolutely essential. This is not important in the first few lines of a program, which are probably read only once by the micro. REMs in lines that are read many times (especially in a FOR...NEXT loop) can slow a program down considerably. The same applies to REMs on the first line of a frequently-called subroutine (another reason for putting them on the line above).

(2) Define the most commonly used variables first in the program, before the less commonly used ones. The reason for this is that the micro stores the values of each variable in a table in RAM. It stores the variables in the order in which they first appear in the program. Every time it has to find out the value of a variable, or to change it, it goes to the table and looks through the list to find the variable. Those variables which are near the top of the list are found more

quickly than those further down the list. This is especially important for variables which are used in loops and may have to be evaluated thousands of times during the running of a program. It is easy to declare such variables early in the program, even if they are not actually being used at that stage, by including a line such as:

```
10 N = 0:X = 0:P = 0
```

In this line, N would be the most frequently used variable of all. As with many of these time-saving techniques, they are only worth bothering about if you must make the program run at maximum speed, or if you have a long program with many variables in use. (3) Use variables, not numerical values, in loops. To see what difference this makes, consider the example in Listing 12.5. It takes

```
10 FOR J = 1 TO 100
20 PRINT 1234 * 5678
30 NEXT J
```

Listing 12.5. A loop in which the VIC has to convert the numbers before multiplying them.

3.45 seconds to run. Each time through the loop, the VIC has to convert the figures '1234' and '5678' into their equivalents in its binary codes, before it can get on with the business of multiplying them together and PRINTing the answer. The program in Listing 12.6, which produces the same result, takes only 2.82 seconds to run – a reduction of 18%.

```
5 X = 1234: Y = 5678
10 FOR J = 1 TO 100
20 PRINT X*Y
30 NEXT J
```

Listing 12.6. Here the conversion is done before the loop begins.

The two numbers are evaluated once and for all in line 5. At line 20, the VIC merely has to look up the values of X and Y in its variable table before working out the result. Looking up values in a variable table is quicker than evaluating the numbers afresh each time around the loop.

Interacting

The user interacts with the computer in two ways – when typing something on the keyboard and when reading something on the TV screen. Your programs should make it easy for the user to

communicate with the computer through the keyboard and for the computer to communicate with the user by way of the screen.

Many of the people who want to use computers, and like to use computers, are not computer experts and are not typing experts either, so it is essential to make the use of the keyboard as simple and effortless as possible. First of all, the user must be told clearly what he or she is expected to do at each stage. Instructions can take the form of a manual, a printed or written sheet, or a display on the screen. With the unexpanded VIC there is not usually memory to spare for storing screenfuls of explanatory text, but even a few bytes of memory can be put to good effect. Examples of helpful INPUT messages are:

```
Do you want to play again (Y/N)?
How many players (1 -4)?
Which month (1 - 12)?
Press any key to continue
```

The first example indicates to the user that the expected answer is either 'Yes' or 'No' and that the correct response is to press either key 'Y' or key 'N'. It is unreasonable to expect the user to type the whole words 'Yes' or 'No' when single letters are perfectly clear. The second example tells the user what range of numbers is permitted. The third example specifies the range and informs the user that the month should be typed in as a number, not as a word. Accepting numbers instead of words helps the user. It is expecting too much for a non-typist to type 'Febraury', sorry, 'February', when the number '2' will do.

The VIC has both INPUT and GET (or GET$) as a way of receiving data from the user. GET and GET$ can be used only when a single key is to be pressed. You could use GET$ to receive 'Y' or 'N', any number in the range Ø to 9, or 'any key' in the fourth example above. GET$ is also used in games for firing guns, moving bats and many other operations, often in conjunction with the function keys.

It is best not to use GET (or GET$) and INPUT in the same program. INPUT requires the RETURN key to be pressed, while GET and GET$ do not. This is likely to confuse the user. If you must use both in the same program, display the instruction to press RETURN every time you use INPUT. Similarly, if an INPUT statement requires several numbers, tell the user how many numbers are to be entered, and remind the user to separate them by commas. The micro's own response to mistaken entries, '? REDO FROM START', is not encouraging.

Input checking

Even with help, the user may type in the wrong figures. Usually this is unintentional, but there are those people who like to type in ridiculous responses to see if the program can cope with them. Input checking helps to avoid program crashes being caused accidentally or deliberately. If a number has to be typed in and must be in a given range, it is easy to check it (see Listing 12.7).

```
200 PRINT"HOW MANY PLAYERS (1-4)":INPUT
N
210 IF N < 1 OR N >4 THEN PRINT"PLEASE E
NTER A NUMBER BETWEEN 1 AND 4":GOTO 200
```

Listing 12.7. Checking INPUT to make sure it is within the required range.

Note the polite request – a micro has infinite patience and can afford to be polite. Note also how the message in line 21ϕ gives more explicit instructions as to what the correct response should be. Another point to note is that the message requesting information is PRINTed. Short messages can be included as part of the INPUT statement, but the VIC cannot accept INPUT messages more than 20 characters long. Longer messages cause an error, or part of the message may mysteriously appear as part of the answer string.

If you object that your program has many INPUTs and that to provide a polite error message at each INPUT uses too much memory, try to use the same message for each INPUT. At the beginning of the program put the line:

```
1ϕ M$ = "PLEASE ENTER A NUMBER BETWEEN"
```

Then, the line after the INPUT at line 2ϕϕ can be:

```
21ϕ IF N <ϕ OR N> 4 THEN PRINT M$;" 1 AND 4":GOTO
   2ϕϕ
```

You can use different numbers for the range on each occurrence of INPUT.

Similar checking can be used with string inputs. A 'Yes/No' query can be handled as shown in Listing 12.8. The query is PRINTed first, as it is too long to work with INPUT. In this example, the game ends unless the user types 'Y'. There is no point in *forcing* the user to type

```
300 PRINT"DO YOU WANT TO PLAY    AGAIN":I
NPUT A$
310 IF A$ = "Y" THEN 20
320 END
```

Listing 12.8. Checking string INPUTs.

either 'Y' or 'N'. After all, little is lost if the user types 'T' instead of 'Y', for the game can easily be restarted by RUNning it. If it is essential to confirm a negative response, line 32ϕ in Listing 12.8 may be altered as shown in Listing 12.9.

```
320 IF A$ <> "N" THEN PRINT"PLEASE TYPE
'Y' OR 'N'":GOTO 300
```

Listing 12.9. Confirming a negative response.

Sometimes it is important to make the user think carefully before making a response which has drastic consequences (see Listing 12.10). The second INPUT (line 41ϕ) has a short message, so this does not have to be PRINTed separately. If the user responds *twice* with 'Y', it may be taken that it is safe to go ahead and delete all the data. Otherwise the program, goes on to lines 43ϕ and beyond, with the data intact.

```
400 PRINT"DO YOU WANT ALL THE   DATA DEL
ETED (Y/N)":INPUT A$
410 IF A$ = "Y" THEN INPUT"ARE YOU SURE"
;A$
420 IF A$ = "Y" THEN GOSUB 1000: REM TO
DELETE DATA
```

Listing 12.10. Giving the user a chance for a change of mind.

One of the problems with INPUT is that, if it is expecting a number and is given a letter, it will not accept it. The 'REDO FROM START' message appears on the screen. Yet it is fairly easy for a beginner to press one of the top row of letter keys in mistake for the numbers, or to type 'THREE' when the computer is expecting '3'. One way of avoiding this kind of error is to take *all* INPUTs as strings (see Listing 12.11). This routine is slightly longer than the one given earlier. VAL(A$) returns a value zero for alphabetic entries. The routine asks for a new entry if a letter or letters are typed or if a number is typed which is out of range.

```
500 INPUT"WHICH MONTH (1-12)";A$
510 A = VAL(A$)
520 IF A<1 OR A>12 THEN PRINT"PLEASE TYP
E A NUMBER  BETWEEN 1 AND 12":GOTO 500
```

Listing 12.11. Using string INPUTs for checking numbers.

The only trouble with checking input is that the routines for the checking can easily occupy more program lines than the active part of the program. In mathematical programs, in which it is essential that data is thoroughly checked before it is worked upon, the input routines may make up as much as three quarters of the program. It is

often a matter of compromising between thorough input checking, on the one hand, and keeping the program to a manageable length, on the other.

The utility programs in this book lack any kind of input checking, so please do not take these as examples to follow! The aim has been to save you work when you are first keying in these programs. Assuming that you now have these utilities on tape, this might be a good opportunity to apply the advice given above and add some input routines of your own.

Output

The micro is able to communicate with the user either through the screen or by means of the sound generators. It is easy to program the VIC to produce a sound to attract the user's attention. Preferably, this should be a musical tone, or series of tones. The tones used before a public announcement at an airport are an example. This is much to be preferred to the peremptory 'beep' that certain computers emit!

Colour can play its part in guiding the user. In the Newkeys and Multikeys utilities (Listings 7.3 and 7.4), the colour of the border changes to green to indicate that the key design has been accepted and the codes are being calculated. This is preferable to having an uneasy pause while the user wonders if anything is happening or if, perhaps, the wrong key has been pressed. In other programs which involve lengthy calculations it is helpful to put a message such as 'CALCULATING – PLEASE WAIT' on the screen. The user then knows that *something* is happening and does not worry that the program has 'hung-up' for some unknown reason.

Another way in which colour can help is by indicating which stage of the program has been reached. At one stage the screen may be pink, with a blue border and red letters. At another stage other colours may be used. The colouring helps the user to know what stage has been reached and what kinds of response are expected. Colour can help draw attention to headings or to important messages which are displayed in a colour different from that of the other text. With such a versatile computer as the VIC there is no excuse for dull and uninteresting displays.

To summarise

In this chapter, you have found out how to:

- use (or *not* use) REM
- document your programs for reference by others and yourself
- economise on memory space
- make your programs run as fast as possible
- handle input from the keyboard
- make your friendly VIC even friendlier.

Appendix A
Useful Tables

Table 1 What the controlling keys do
These keys have the same action in *both* modes

Key	*What it does*
CLR/HOME	Sends the cursor home; with SHIFT, it clears the screen too.
CRSR (with arrows)	Move cursor down and right; with SHIFT, they move it up and left.
CTRL	Changes colours (see Table 2); also makes a program run more slowly.
'Flag' (Commodore sign)	Gives left side symbols (see Table 2). Used with SHIFT it changes mode. Can be used instead of SHIFT for controlling the cursor.
INST/DEL	Deletes characters; use with SHIFT to insert characters.
RETURN	Used at the end of an instruction or line of a program.
RESTORE	Press with RUN/STOP to clear the screen and restore VIC to its power-on state.
RUN/STOP	Stops the VIC when it is running a program (see page 16). Also used with RESTORE (see above). Used with SHIFT, it gives the command to LOAD a program from tape.
SHIFT	Changes the action of many of the keys (see above and Table 2). Used with 'Flag', it changes mode.
SHIFT LOCK	Press and lock down, to give same effect as holding SHIFT down. Press again to release.

Table 2 What the other keys do - graphics mode

Key	*When used alone*	*What it does With SHIFT*	*With 'Flag'*	*With CTRL*
A to Z	A to Z	Right side set	Left side set	—
1 to 8	1 to 8	Top symbols*	Top symbols*	Change colour
9	9	)	)	Reverse on
∅(zero)	∅	∅	∅	Reverse off
+ − £ @ *	+ − £ @ *	Right side set	Left side set	—
: ; , . /	: ; , . /	[ ] < > ?	[ ] < > ?	—
↑	↑	π	π	—
= ←	= ←	= ←	= ←	—
Space-bar	space	space	space	—

* The symbols marked on the keys above the number.
The 4 light brown function keys, to the right of the main keyboard are described in Chapter Ten.

Table 3 What the other keys do - text mode

Key	*When used alone*	*What it does With SHIFT*	*With 'Flag'*
A to Z	a to z	A to Z	Left side set
+ −	+ −	Right side set	Left side set
£	£		Left side set
@	@	✓	Left side set
*	*	Right side set	
↑	↑		

Keys not listed above do the same as in Graphics Mode (see Table 2). Keys in this table do nothing when pressed with CTRL.

Table 4 Doing what you want

Action	*What to do*
Send cursor home	Press CLR/HOME
Send cursor home and clear screen	Press CLR/HOME with SHIFT
Move cursor down or right	Use CRSR keys
Move cursor up or left	Use CRSR keys with SHIFT
Change colour	Use number keys 1 to 8, with CTRL
Inverse video on	Press key 9 with CTRL
Inverse video off	Press key ∅ (zero) with CTRL
Delete character	Press INST/DEL (hold to delete more)
Change mode	Press SHIFT with 'Flag' (Commodore key)
Write in capital letters	In Graphics Mode, use alphabet keys without SHIFT *OR* in Text Mode, use alphabet keys with SHIFT
Write in lower-case (small letters)	Change to Text Mode, and use alphabet without SHIFT
Write top symbols	Press SHIFT with the key
Write left side set	Press 'Flag' with the key (* depends on mode, see Table 3)
Write right side set on alphabet keys	Change to Graphics Mode and press SHIFT with key
Write right side set on + − £ @ * keys	Press SHIFT with the key (£ and @ depend on Mode, see Table 3)
Restore VIC for a fresh start	Press RUN/STOP with RESTORE

Table 5 Setting screen and border colours

The values in the body of the table are those to be POKEd to address 36879 to set screen and border colours. To help you relate these to codes for other colour graphics the colour codes for each colour are given in brackets after the name of each colour.

	Border colours							
Screen Colours	*Black* (ϕ)	*White* (1)	*Red* (2)	*Cyan* (3)	*Purple* (4)	*Green* (5)	*Blue* (6)	*Yellow* (7)
Black (ϕ)	8	9	10	11	12	13	14	15
White (1)	24	25	26	27	28	29	30	31
Red (2)	40	41	42	43	44	45	46	47
Cyan (3)	56	57	58	59	60	61	62	63
Purple (4)	72	73	74	75	76	77	78	79
Green (5)	88	89	90	91	92	93	94	95
Blue (6)	104	105	106	107	108	109	110	111
Yellow (7)	120	121	122	123	124	125	126	127
Orange (8)	136	137	138	139	140	141	142	143
Light orange (9)	152	153	154	155	156	157	158	159
Pink (10)	168	169	170	171	172	173	174	175
Light cyan (11)	184	185	186	187	188	189	190	191
Light purple (12)	200	201	202	203	204	205	206	207
Light green (13)	216	217	218	219	220	221	222	223
Light blue (14)	232	233	234	235	236	237	238	239
Light yellow (15)	248	249	250	251	252	253	254	255

Note: cyan is a bluish-green colour, sometimes called 'turquoise'.

Table 6 Screen codes for letters and symbols

For some codes, the character printed depends on whether the VIC is in Graphics Mode or in Text Mode. Symbols which appear between Graphics Mode and Text Mode columns are the same for both modes.

Code	*Graphics Mode*	*Text Mode*	*Code*	*Graphics Mode*	*Text Mode*
0	@		24	X	x
1	A	a	25	Y	y
2	B	b	26	Z	z
3	C	c	27	[	
4	D	d	28	£	£
5	E	e	29	]	
6	F	f	30	↑	
7	G	g	31	←	
8	H	h	32	(space)	
9	I	i	33	!	
10	J	j	34	"	
11	K	k	35	#	
12	L	l	36	$	
13	M	m	37	%	
14	N	n	38	&	
15	O	o	39	'	
16	P	p	40	(	
17	Q	q	41	)	
18	R	r	42	*	
19	S	s	43	+	
20	T	t	44	,	
21	U	u	45	−	
22	V	v	46	.	
23	W	w	47	/	

Table 6 (*contd*)

Code	*Graphics Mode*	*Text Mode*	*Code*	*Graphics Mode*	*Text Mode*
48		∅	76		L
49		1	77		M
50		2	78		N
51		3	79		O
52		4	80		P
53		5	81		Q
54		6	82		R
55		7	83		S
56		8	84		T
57		9	85		U
58		:	86		V
59		;	87		W
60		<	88		X
61		=	89		Y
62		>	90		Z
63		?	91		
64			92		
65		A	93		
66		B	94		
67		C	95		
68		D	96		(space)
69		E	97		
70		F	98		
71		G	99		
72		H	100		
73		I	101		
74		J	102		
75		K	103		

Table 6 (*contd*)

Code	*Graphics Mode*	*Text Mode*	*Code*	*Graphics Mode*	*Text Mode*
104			116		
105			117		
106			118		
107			119		
108			120		
109			121		
110			122		
111			123		
112			124		
113			125		
114			126		
115			127		

Table 7 The colour codes for displayed characters

Colour	*Code*
Black	0
White	1
Red	2
Cyan (Turquoise)	3
Purple	4
Green	5
Blue	6
Yellow	7

Table 8 VIC 20 sound generators

The table shows how to calculate the frequency of the tone generators (see also Table 9). In general terms, 128 produces the lowest frequency while 254 produces the highest.

Name	*Address*	*Range of values*	*Effect of POKEing value N*
Bass tone	36874	128 to 254	Frequency = 8192/(255 − N)
Tenor tone	36875	128 to 254	Frequency =16384/(255 − N)
Treble tone	36876	128 to 254	Frequency = 32768/(255 − N)
Noise	36877	128 to 254	Low rumble when N = 128 Hissing when N = 254
Volume	36878	0 to 15	Silence when N = 0 Loudest when N = 15

Table 9 Notes of the musical scale

These are the values to be POKEd to the tone generators. The unnamed notes are the semitones between the notes named in the lines above and below.

Note	Treble	Tenor	Bass
F"	232		
E"	230		
	229		
D"	227		
	226		
C"	224		
B'	222		
	220		
A'	218		
	216		
G'	214		
	211		
F'	209	232	
E'	206	230	
	203	229	
D'	200	227	
	196	226	
C'	193	224	
B	189	222	
	185	220	
A	181	218	
	177	216	
G	172	214	
	167	211	
F	162	209	232
E	157	206	230
	151	203	229
D	147	200	227
	138	196	226
Middle C	131	193	224
B		189	222
		185	220
A		181	218
		177	216
G		172	214
		167	211
F		162	209

Table 9 (*contd*)

Note	Treble	Tenor	Bass
E		157	206
		151	203
D		147	200
		138	196
C		131	193
B			189
			185
A			181
			177
G			172
			167
F			162
E			157
			151
D			147
			138
C			131

Table 10 The ASCII Codes

This table lists those codes which are implemented in the standard way on the VIC.

Code	*char*	*Code*	*char*
13	(Return)	61	=
32	(Space)	62	>
33	!	63	?
34	"	64	@
35	#	65	A
36	$	66	B
37	%	67	C
38	&	68	D
39	'	69	E
40	(	70	F
41	)	71	G
42	*	72	H
43	+	73	I
44	,	74	J
45	-	75	K
46	.	76	L
47	/	77	M
48	0	78	N
49	1	79	O
50	2	80	P
51	3	81	Q
52	4	82	R
53	5	83	S
54	6	84	T
55	7	85	U
56	8	86	V
57	9	87	W
58	:	88	X
59	;	89	Y
60	<	90	Z

Table 11 The function keys and their CHR$ codes

Unshifted/ shifted	*Key*	*Number*	*CHR$ code*
Unshifted	Top	F1	133
	2nd down	F3	134
	3rd down	F5	135
	Bottom	F7	136
Shifted	Top	F2	137
	2nd down	F4	138
	3rd down	F6	139
	Bottom	F8	140

Table 12 Programming moving user-defined graphics (Method 2)

No of chars	*Define as keys*	*Line 5 reads* 'FOR J=∅ TO...'	*Line 7 reads* 'POKE...+J,X'
1	Z	7	6352
2	YZ	15	6344
3	XYZ	23	6336
4	WXYZ	31	6328
5	VWXYZ	39	6320
6	UVWXYZ	47	6312
N	last N keys	8*N − 1	6360 − 8*N

Table 13 Programming moving user-defined graphics (Method 3)

No of chars	*Define as keys*	*Line 5 reads* 'FOR J=∅ TO...'	*Line 7 reads* 'POKE...+J,X'
1	Z	7	7376
2	YZ	15	7368
3	XYZ	23	7360
4	WXYZ	31	7352
5	VWXYZ	39	7344
6	UVWXYZ	47	7336
N	last N keys	8*N − 1	7384 − 8*N

Appendix B

Converting Programs to Larger RAM

The programs in this book have been written for the unexpanded VIC 20, which has approximately 3.5 kilobytes of user RAM. If you have additional RAM it may be necessary for you to alter one or two values in a few of the programs.

If you have added 3K (3 kilobytes) of extra RAM (and no more than that) the programs do not need to be modified. You need read no further.

If you have added more than 3K of extra RAM, the VIC locates its character code RAM and colour code RAM at new addresses. This means that those graphics programs in which addresses are POKEd with character codes or colour codes will have to be modified to take account of the new addresses. Programs which do not POKE to these areas of RAM (and this means most of the programs in the book) do *not* need to be modified.

The changes of address are:

Character code RAM now begins at address 4096 (instead of 7680).

Colour code RAM now begins at address 37888 (instead of 38400).

There are 3 kinds of change to make:

(1) When POKEing characters to character code RAM, reduce all addresses by 3584, so that they now run from 4096.
(2) When POKEing colours to colour code RAM, reduce all addresses by 512, so that they now run from 37888.
(3) Some programs use the value 30720 for calculating colour code RAM addresses from the corresponding address in character code RAM; alter 30720 to 33792 in these programs.

Index

addition, 34, 35, 46
address, 21
Agequiz program, 74–6
Air Raid program, 54–6
alphabetical sort, 108–110
AND operator, 75, 91
Animalogic program, 78
Animals program, 76–7
animation, 84, 138, 142, 154, 156
array, 57–61
ASCII code, 81–2, 91, 93, 129, 180
ASCII function, 91
ATN function, 131
auxiliary colour, 94, 98, 99, 150, 151, 156

background colour, 94, 99, 150, 153
binary code, 86, 94–7
blackboard routine, 19–20
Boolean logic, 78
border colour, 14, 94, 99, 150, 153, 173
bubble sort, 114–16

Carousel program, 61–7
cartridge, 6
character code RAM, 23–4, 28, 182
character generator, 85
CHR$ function, 81–2, 106, 129, 181
circle drawing program, 131–33
Clock program, 103
CLR/HOME key, 14, 170
code, ASCII, 81–2, 91, 93, 129, 180
  binary, 86
  character, 23–4, 174–6
  colour, 25, 44, 177
  control, 31, 66, 82, 84
collisions, 147–8
colour code, 25, 44, 177
colour code RAM, 25–6, 148–9, 182
colours, 16–18, 44, 94, 172, 173
  moving, 148–9
  of screen and border, 21, 168, 173
control character, 83
control code, 31, 66, 82, 84
COS function, 131, 132
CRSR keys, 14, 170
CTRL key, 17, 170
cursor, 14, 172

DATA statement, 56–7, 61
decisions, 74–80
DEF FN statement, 134
defining characters, 88
degrees of angle, 131, 133
delay loop, 51, 155
deletions, 14, 172
DIM statement, 58, 60
division, 35
documentation, 160

elapsed-time program, 73
empty string, 47–8
envelope, 55, 120
exponential form, 39
exponentiation, 37

false, 78
faster programs, 163–4
FGX utility program, 117–26
firmware, 5
'Flag' key, 18, 170
flag variable, 65–6, 107
floating-point variable, 43
flow chart, 72, 107, 160
FN function, 134
FOR...TO...NEXT loop, 28, 69, 161–2
foreground colour, 94, 98, 99, 150, 153
FRE function, 109–110
frequency, 52, 177–9
function keys, 128–30, 181
function, user-defined, 133–7

generator, character, 85

GET statement, 47–8, 128, 165–8
GOSUB statement, 64, 163
GOTO statement, 31
graphics mode, 19, 171

hardware, 4

IF...THEN statement, 48, 74
INPUT statement, 44, 47–8, 51, 165–8
INST/DEL key, 14, 170
INT function, 29, 64
integer, 29, 64
integer variable, 43, 58, 163
interchange sort, 111–14
inverse video, 172

Jellyfish program, 149

keyboard, 13–16
keys, 170

left-side graphics set, 18–19, 171–2
LEFT$ function, 101
LEN function, 105
LET statement, 160–61
LIST command, 30
locomotion, 143
logical operator, 75–9
Longspell program, 105
loops, 69–73

Magic Carpet program, 28–32
memory, 5, 162–3
MID$ function, 102
modulator, 10, 11
monitor program, 4
monitor screen, 4, 7
Monster program, 155–6
motorcycle crash effect, 123–6
moving disc program, 149
multicolour graphics, 94–100, 150–52
Multikeys utility program, 94–100, 139, 150
Multi-Monster program, 156–7
multiplication, 29, 35
multiplication programs, 69–71
multistatement line, 162

nested loops, 71, 107
NEW command, 28
Newkeys utility program, 89–93, 139
noise generator, 50, 121
NOT operator, 75
null string, 47–8
numeric variable, 40–43, 58

ON...GOSUB statement, 79–80
ON...GOTO statement, 79–80
operator, 35
operator, arithmetic, 35
  logical, 75–9
  relational, 74, 161
OR operator, 75

PEEK function, 85, 148
π, 15, 131–3
POKE statement, 21–2, 82, 146
POS function, 129
power supply, 12
PRINT statement, 82–5
priorities, 35–7, 49
program, 4, 27
Puffer program, 82–4
Puppet program, 153–5
Put and Take program, 79

radian, 131, 133
RAM, 6, 21, 90, 182
random number, 29
random number function, 136
range of variables, 39–40, 43
Reaction timer program, 47
READ statement, 56–7
relational operator, 74, 161
REM statement, 158–60, 162, 163
RESTORE key, 16, 170, 172
RESTORE statement, 57
RETURN key, 22, 27, 170
RETURN statement, 64
right-side graphics set, 15–17, 171–2
RIGHT$ function, 101
RND function, 29, 136
ROM, 5, 22
rounding off, 35, 135–6
RUN/STOP key, 16, 170, 172
RVS ON and RVS OFF keys, 18, 83–4, 172

screen, 14
screen code, 23–4, 174–6
scrolling, 24
SHIFT keys, 14–16, 170
SHIFT LOCK key, 170
SIN function, 130, 132
sockets, 9–11
software, 5
sort, alphabetical, 108–110
  bubble, 114–16
  interchange, 111–14

sound generator program, 50
sound generators, 50, 117–19, 177–9
SPC function, 147
special variable, 46
SQR function, 130
square root, 37, 130
standard form, 39
STEP statement, 70
string variable, 45–6, 58
STR$ function, 104
subscript, 58
subtraction, 35, 102
swapping variables, 111–13

Talker program, 140, 142–3
TAN function, 131
tape, 5, 7
Target game program, 129
text mode, 19, 171
TI and TI$, 46–7
timing program, 73
tone generators, 50
Traffic-light program, 27
true, 78
TV set, 6–7, 12–13

UFO program, 150–52
user-defined graphics, 86–101, 139–42, 181
user-friendliness, 165–6, 168
Using π program, 133
utility program, 6

VAL function, 104, 167
variable, 40–47, 163
  flag, 65–6, 107
  floating-point, 43
  integer, 43, 58, 163–4
  numeric, 40–43, 58
  special, 46
  string, 45–6, 58
variable names, 42
Video Interface Chip, 50, 90, 98, 177
video monitor, 4, 7
volume control, 51–3

Walker program, 144–5

+, 35
−, 35
*, 29, 35
/, 35
↑, 37
<, 73, 108
>, 74, 108
=, 74, 161
π, 15, 131–3

THE ORIC-1 PROGRAMMER
M. James and S. M. Gee
0 246 12157 2

ORIC MACHINE CODE HANDBOOK
Paul Kaufman
0 246 12150 5

The TI 99/4A

GET MORE FROM THE TI99/4A
Garry Marshall
0 246 12281 1

The VIC 20

GET MORE FROM THE VIC 20
Owen Bishop
0 246 12148 3

THE VIC 20 GAMES BOOK
Owen Bishop
0 246 12187 4

The ZX Spectrum

THE ZX SPECTRUM And How To Get The Most From It
Ian Sinclair
0 246 12018 5

THE SPECTRUM PROGRAMMER
S. M. Gee
0 246 12025 8

THE SPECTRUM BOOK OF GAMES
M. James, S. M. Gee and K. Ewbank
0 246 12047 9

INTRODUCING SPECTRUM MACHINE CODE
Ian Sinclair
0 246 12082 7

SPECTRUM GRAPHICS AND SOUND
Steve Money
0 246 12192 0

THE ZX SPECTRUM How to Use and Program
Ian Sinclair
0 586 06104 5

Learning is Fun! 40 EDUCATIONAL GAMES FOR THE SPECTRUM
Vince Apps
0 246 12233 1

The ZX81

THE ZX81 How to Use and Program
S. M. Gee and Mike James
0 586 06105 3

Which Computer?

CHOOSING A MICROCOMPUTER
Francis Samish
0 246 12029 0

Languages

COMPUTER LANGUAGES AND THEIR USES
Garry Marshall
0 246 12022 3

EXPLORING FORTH
Owen Bishop
0 246 12188 2

Machine Code

Z-80 MACHINE CODE FOR HUMANS
Alan Tootill and David Barrow
0 246 12031 2

6502 MACHINE CODE FOR HUMANS
Alan Tootill and David Barrow
0 246 12076 2

Using Your Micro

COMPUTING FOR THE HOBBYIST AND SMALL BUSINESS
A. P. Stephenson
0 246 12023 1

DATABASES FOR FUN AND PROFIT
Nigel Freestone
0 246 12032 0

SIMPLE INTERFACING PROJECTS
Owen Bishop
0 246 12026 6

INSIDE YOUR COMPUTER
Ian Sinclair
0 246 12235 8

Programming

THE COMPLETE PROGRAMMER
Mike James
0 246 12015 0

PROGRAMMING WITH GRAPHICS
Garry Marshall
0 246 12021 5

Word Processing

CHOOSING A WORD PROCESSOR
Francis Samish
0 246 12347 8

WORD PROCESSING FOR BEGINNERS
Susan Curran
0 246 12353 2

TAKE COMMAND OF THE VIC 20!

This book is written for the VIC 20 beginner as well as for the user who has already had some experience on this or other machines.

The VIC 20 is essentially a user-friendly computer, with excellent graphics and with exceptional sound capabilities for a machine in its price range. This book shows you how to get the best from these facilities. If you simply want to key in a short routine, and then watch the dancing colours and listen to the sounds, there is plenty to amuse you here. On the other hand, if you want to move on quickly to writing your own programs, this book will show you how. As well as exploring the more spectacular aspects of the VIC 20, this book takes you further along the road to good programming. It explains in detail and with plenty of examples how to use the main commands of the VIC 20 BASIC and also supplies you with a host of routines which provide reliable and effective short-cuts to competent programming.

The Author

Owen Bishop is a freelance technical writer and programmer. He is the author of about thirty books, including a number on popular computing. He is a well known and regular contributor to computing journals.

Also from Granada

THE VIC 20 GAMES BOOK
21 Challenging Games
Owen Bishop
0 246 12187 4

INSIDE YOUR COMPUTER
Ian Sinclair
0 246 12235 8

PROGRAMMING WITH GRAPHICS
Garry Marshall
0 246 12021 5

THE COMPLETE PROGRAMMER
Mike James
0 246 12015 0

GRANADA PUBLISHING
Printed in Great Britain 0 246 12148 3

£5.95 net